AFIT/GE/ENG/91D-38

AN INVESTIGATION OF A DESIGN FOR A FINITE-DIFFERENCE TIME DOMAIN (FDTD) HARDWARE ACCELERATOR

THESIS

James Raley Marek
Captain, USAF

AFIT/GE/ENG/91D-38

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1991	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE AN INVESTIGATION OF A DESIGN FOR A FINITE-DIFFERENCE TIME DOMAIN (FDTD) HARDWARE ACCELERATOR	5. FUNDING NUMBERS
6. AUTHOR(S) James Raley Marek	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GE/ENG/91D-38
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) WL/AARA-2 (Capt Pierre LeFevre) Avionics Directorate, Wright Laboratory WPAFB OH 45433	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This study investigated increasing the speed of Finite-Difference Time Domain (FDTD) cell calculations through a special purpose architecture using Very Large Scale Integration (VLSI). These FDTD cell equations model inhomogeneous, isotropic, lossy magnetic and dielectric materials. Special attention was given to simplicity and performance, using the fastest components generally available in AFIT VLSI programs, while attempting to minimize component count. A VHSIC Hardware Description Language simulation of the proposed chip established design feasibility and provided performance estimates: 350 ns to generate the first cell value, 200 ns thereafter (30 MFLOPS maximum double-precision).

This study also implemented boundary conditions in hardware as well. No new hardware was designed; instead, the algorithm was translated into microcode for use by the AFIT Floating Point Application Specific Processor. The first boundary value is computed in 850 ns, with successive results calculated every 300 ns thereafter (43 MFLOPS maximum double-precision).

Standard FDTD FORTRAN codes were run on a SPARC2 workstation and execution times compared to modified codes simulating the implementation of the above hardware. On a 66 cubic cell free-space computational domain, these chips reduced total FDTD code execution time by a factor of 4.9, and cell and boundary calculation time by a factor of 9.5.

14. SUBJECT TERMS FDTD, VLSI, Vector Processing, Microprogramming, Finite-Difference Time Domain, Electromagnetic Fields, Computer Architecture, Very Large Scale Integration	15. NUMBER OF PAGES 118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

AFIT/GE/ENG/91D-38

AN INVESTIGATION OF A DESIGN FOR A FINITE-DIFFERENCE TIME DOMAIN (FDTD) HARDWARE ACCELERATOR

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

James Raley Marek, B.S.E.E.

Captain, USAF

December 1991

Approved for public release; distribution unlimited

Preface

This study grew out of an interest to try to use dedicated computer architectures to speed up computational electromagnetics calculations. At first, an attempt was made at speeding up moment method calculations. It soon became clear, however, that working with the finite-difference time domain equations would be a simpler task, and as time was growing short, simplicity became the rule.

My thanks go to Dr. Raymond Luebbers of Pennsylvania St. University and Dr. Edward Newman of Ohio St. University, both of whom provided guidance and insight into the world of computational electromagnetics. I give special thanks to Dr. Andrew Terzuoli, my faculty advisor, and Capt Mark Mehalic, who both supported me in my quest for a mix of electromagnetics and computer architecture, as well as Maj Harry Barksdale and Maj Bill Hobart for their encouragement and expertise. I also wish to thank the all the members of my committee, including LTC Baker and Dr. Lamont.

Finally, I thank my wife, Candy, who never ceases to amaze me and whom I admire more each day, and who brought forth our beautiful daughter Ariece, the only person who can reset my computer and bring a smile to my face at the same time. Most importantly, I thank God for the blessings of a wonderful family, for the abilities that I possess, and for the strength and guidance he has given me.

Table of Contents

List of Figures

List of Tables

Abstract

This study investigated increasing the speed of Finite-Difference Time Domain (FDTD) cell calculations through a special purpose architecture using Very Large Scale Integration (VLSI). These equations model inhomogeneous, isotropic, lossy magnetic and dielectric FDTD problems. Special attention was given to simplicity and performance, using the fastest components generally available in AFIT VLSI programs, while attempting to minimize component count. A VHSIC Hardware Description Language simulation of the proposed chip established design feasibility and provided performance estimates: 350 ns to generate the first cell value, 200 ns thereafter (30 MFLOPS maximum double-precision).

This study also implemented boundary conditions in hardware as well. No new hardware was designed; instead, the algorithm was translated into microcode for use by the AFIT Floating-Point Application Specific Processor. The first boundary value is computed in 850 ns, with successive results following every 300 ns (43 MFLOPS maximum double-precision).

Execution times of standard FDTD FORTRAN codes run on a SPARC2 workstation were compared to those of modified codes simulating the implementation of the above hardware. On a 66 cubic cell free-space computational domain, these chips reduced total FDTD code execution time by a factor of 4.9, and cell and boundary calculation time by a factor of 9.5.

HARDWARE IMPLEMENTATION
OF THE
FINITE-DIFFERENCE TIME DOMAIN EQUATIONS

I. Background

Introduction

In general, the solution to Maxwell's electromagnetic equations in the presence of a scatterer cannot be written in closed form and must be approximated. With the advent of the electronic computer, researchers and engineers today can study and compute the scattering from complex, yet small, shapes. However, computers still do not possess the power necessary to determine the scattering from large, complex objects. Various ray tracing methods exist for studying these larger problems, but generally their solution error is much higher than many research programs can tolerate.

When high accuracy is desired, one must usually consider numerical approximations of the solution to Maxwell's equations. Among these are variational techniques, moment methods, and time domain methods. The problem with these methods, however, is that they require large amounts of memory and large numbers of floating-point (real and complex) computations in order to determine a result. Therefore, these techniques are effectively limited to objects on the order of tens of wavelengths or less. When engineers need to exceed these limits, they pay the price in time spent waiting for results. Even with an eight processor Cray Y-MP/8 supercomputer, researchers have calculated the electromagnetic scattering from structures only the size of aircraft engine intakes (1:16-19). Moreover, the dollar cost of even this limited capability can easily exceed most research budgets.

Since the U.S. Air Force is interested in the scattering from entire aircraft, an alternative approach is necessary if they are to find accurate solutions to such problems in reasonable amounts of time and at a reasonable cost. Standard sequential computer architectures are presently too slow to efficiently solve the large scattering problems. Current research is attempting to find efficient solutions to these problems through the use of general-purpose, parallel and vector computer architectures, as well as through specialized hardware but, as yet, no one has attempted to develop a simple, inexpensive, high performance architecture committed solely to computing electromagnetic fields. A specialized, high-speed computer architecture, when produced in large numbers and operating in parallel, may be able to significantly decrease the time required for these field calculations, and do so at a lower cost.

Problem Statement

The objective of this study was to speed up electromagnetic scattering calculations through the use of Very Large Scale Integration (VLSI) technology. Specifically, this study presents the design of a circuit that rapidly computes the cell field values of the finite-difference time domain (FDTD) method and investigates how such a circuit might improve the run times of FDTD computer programs. Furthermore, the possibilities of speeding up the calculation of the FDTD radiation boundary condition is also explored.

Scope

Due to the constrained time frame of this thesis effort, this study is limited to the following:

1. Development of a specification for computational circuitry that embodies the Yee equations of the FDTD method (2:303). This specification takes the form of Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) files, which model the FDTD circuit down to at least the functional unit level.

2. Investigation of computational circuitry which might speed up the calculation of the FDTD radiation boundary condition equations.

This study did not attempt to resolve any of the limitations of the FDTD method, nor did it result in the actual construction of any hardware.

Assumptions

In order to further simplify this study, the following assumptions have been made:

1. Interface circuitry shall be designed at a later time. This permits design decisions free from the constraints of external interface requirements, and leaves the interface design to be studied as a completely separate issue.

2. The external world is able to supply the FDTD chip with data at 40 MHz clock rates. This allows greater simplification in the analysis of the maximum possible speed-up achievable due to the operation of the circuit. This is not an unrealistic assumption, since several vendors

are already producing 20-ns, 1-Mbit static random access memory (RAM) chips for under $100 with 4-Mbit RAM chips of the same speed soon to come (3:99-105).

3. Input and output conform to the IEEE 754-1985 64-bit floating-point representation (double-precision) for numbers (4). This is a well-accepted standard and provides for a 53-bit mantissa.

Approach

The cell and boundary condition equations of the FDTD method are the foundation upon which this design is based. The tasks accomplished in this study are as follows:

1. Studied and manipulated both cell and boundary equations to improve computational efficiency.

2. Developed a data sequence diagram describing an FDTD cell equation evaluator. This diagram was based directly upon the equations and provides a graphic reference for the following work.

3. Implemented this proposed architecture in VHDL code. All of the functional blocks in the data sequence diagram have behavioral or structural descriptions in VHDL code.

4. Ran VHDL simulations of this code. These simulations helped validate the design of the FDTD chip, and also provided data for performance analysis.

5. Wrote Floating-Point Application Specific Processor (FPASP) microcode for calculating the FDTD boundary values. Ran simulations of the FPASP in VHDL using this microcode to validate the correctness of the microcode and determine computation times.

6. Ran simulations of existing FDTD FORTRAN code as well as codes modified to reflect the presence of the above designs. Reported on the performance of the design. Stated probable performance effects on FDTD code run times. Attempted to determine cost/performance ratio and relate to the that of present designs.

7. Studied and reported on likely connection networks and data communication needs in a parallel application of this hardware.

It should be noted that several of the above steps will involve iteration and trade-off analysis. Throughout this entire effort, issues and decisions shall be appropriately documented so that the basis of the final design can be readily understood.

II. Current Efforts

General Purpose Parallel Architectures

Recent attempts to reduce computation time through parallelization of the FDTD method have focused primarily on general purpose machines. Among those most often reported in the technical literature are hypercubes, a relatively large-grained, distributed-memory computer architecture, the Connection Machine, an extremely fine-grained (64K 1-bit processors), distributed-memory computer architecture, and the Cray Y-MP/8, a large grained, shared-memory vector computer architecture.

The FDTD method chops the volume of space where scatterer and the unknown fields exist into very small, identical cubes. Each cube in a plane of the volume, as implemented by Perlik on the Connection Machine (5:2912), or a sub-volume of cubes, as implemented by Calalo on the hypercube (6:2900), is assigned to a specific processor in a parallel computer. Since waves fields travel through space along continuous paths and do not jump around, the processors rely only on nearest neighbor communication to pass on information concerning the traveling fields.

It would appear that the Connection Machine might outperform the hypercube architecture simply because it is working on some 65,000 cells in parallel while the hypercube is working on only 32. The Connection Machine is handicapped by its one bit-at-a-time processing capability and the need for a great deal of inter-processor communication, but its shear "mass" still enables it to perform significantly faster. One must also note that the Connection Machine studied in the literature was equipped with the optional 32-bit floating-point coprocessors, which were reported to improve run time by a factor of ten (5:2911).

Although the specific algorithm details are lacking, the reports indicate a Connection Machine is capable of processing a 2.4+ million cell volume in about 1.7 seconds per simulation time step (5:2912), while a 32 node hypercube calculates a 2.0+ million cell volume at around 15 seconds per simulation time step (6: 2900). It should be noted that both cell capacities stated above appear tc be the maximum that each machine was capable of supporting in machine memory alone.

Daniel Katz and Allen Taflove reported some (comparatively) stunning computation times on a CRAY Y-MP/8. This eight processor supercomputer ran 1800 time steps through a 3,886,920 cell volume in 3 minutes, 40 seconds (a reported computation rate of 1.6 GFLOPS), or about 0.12 seconds per time step. The problem involved computing the fields propagating inside a 25.4 wavelength (30 inches at 10 GHz) serpentine jet engine duct. Although it is not stated, the figure depicting the problem in the report appears to display a stair-step representation to curvature, suggesting that their FDTD lattice only approximated the smooth duct. The report also states that work is progressing on 30 wavelength structures, automatic mesh generation, subcell models for fine-grained structural features, and higher-order algorithms (1:16-19).

In each of these reports, the scatterer is on the order of tens of wavelengths or less. Somewhat surprisingly, no reports could be found of any implementation of electromagnetic scattering code on the nCUBE-2, a fine-grained, distributed-memory architecture consisting of up to 8,192 64-bit processors (each the equivalent of a VAX 8650), with up to 32 Mbytes of memory per processor. It is touted by its maker as "the fastest super computer for science," possessing a maximum computation rate of 27 GFLOPS (7,8). The nCUBE promises not only faster solutions, but its significant memory should allow larger problems to be solved completely within the machine, without the need for external storage of intermediate results. Assuming only one-forth of its memory is available for the storage of the FDTD data structures, the high end nCUBE computer could compute (in memory) a one billion cell structure (a 10' by 10' by 10' computational domain at 10 GHz, one million cubic feet at 1 GHz). Of course, this performance is not without its price: $250,000 for a 64 node machine, $23 million for the 8,192 node version (8). Still, its maker claims its systems "deliver cost/performance advantages 50 times better than traditional supercomputers" (7).

FDTD Specific Parallel Architectures

Researchers are also attempting to improve the computation speed of the finite-difference time domain method through the use of specific computer architectures designed exclusively for the FDTD method.

Researchers at Electro Magnetic Applications, Inc. (Denver, Colorado) report on a study of a parallel, pipelined architecture with the capability to calculate (in parallel) the six electric and magnetic field components required by the FDTD method. Since the architecture is pipelined, results are generated every clock cycle after the pipe fills (9:2913-2915). This report confirms ideas that were intended to be a part of this thesis study, but for reasons discussed later, it was decided that an architecture of this type did not suit current and near future needs. The paper also discusses a normalization technique designed to reduce the number of multiplications in computing the E and H fields and an apparently new technique for modelling thin wires (9:2915). Although this paper discusses a different (computationally faster) expression for the first-order Mur radiation boundary condition, even the second-order Mur equation is ineffective in some classes of problems, so there seems to be little practical use of this particular formulation.

Wavetracer (Acton, Massachusetts) is marketing a massively parallel (between 4,096 and 16,384 single bit processing elements), single instruction multiple data (SIMD) computer capable of handling million-cell FDTD problems at 0.85 seconds per time step, with an apparent maximum of a 4 million cell problem space (10). This computer reportedly sells for under $100,000 for the smaller model and just over $400,000 for the largest (11). The machine makes use of parallel data input/output to achieve bandwidths of 1 Gbyte/second. This appears to be a Connection Machine with more memory (up to 32K per node) and a specialized three-dimensional connection scheme (10). This computer seems to possess some of the best price/performance numbers of all the machines reported in the literature. It is slower than the Cray by only a factor of seven, yet less expensive by a factor of 70. However, the Wavetracer's maximum problem size is smaller than that of the Cray computer.

III. The FDTD Algorithm

General

The Finite Difference Time Domain method is a discretization of the Maxwell Equations in differential form (curl equations). Starting with Maxwell's equations:

$$\nabla\times H=\varepsilon\frac{\partial E}{\partial t}+\sigma_e E \tag{1}$$

$$\nabla\times E=-\mu\frac{\partial H}{\partial t}-\sigma_m H \tag{2}$$

where μ is the magnetic permeability, ε is the dielectric permittivity, σ_e is the total equivalent conductivity giving rise to electric dissipative currents, and σ_m is the corresponding parameter giving rise to magnetic dissipative currents (12:684, 13:77, 14:27-28). All parameters are real. These equations are separated according to their vector components into a scalar form:

$$\frac{\partial H_z}{\partial y}-\frac{\partial H_y}{\partial z}=\varepsilon\frac{\partial E_x}{\partial t}+\sigma_e E_x \tag{3}$$

$$\frac{\partial H_x}{\partial z}-\frac{\partial H_z}{\partial x}=\varepsilon\frac{\partial E_y}{\partial t}+\sigma_e E_y \tag{4}$$

$$\frac{\partial H_y}{\partial x}-\frac{\partial H_x}{\partial y}=\varepsilon\frac{\partial E_z}{\partial t}+\sigma_e E_z \tag{5}$$

$$\frac{\partial E_z}{\partial y}-\frac{\partial E_y}{\partial z}=-\mu\frac{\partial H_x}{\partial t}-\sigma_m H_x \tag{6}$$

$$\frac{\partial E_x}{\partial z}-\frac{\partial E_z}{\partial x}=-\mu\frac{\partial H_y}{\partial t}-\sigma_m H_y \tag{7}$$

$$\frac{\partial E_y}{\partial x}-\frac{\partial E_x}{\partial y}=-\mu\frac{\partial H_z}{\partial t}-\sigma_m H_z \tag{8}$$

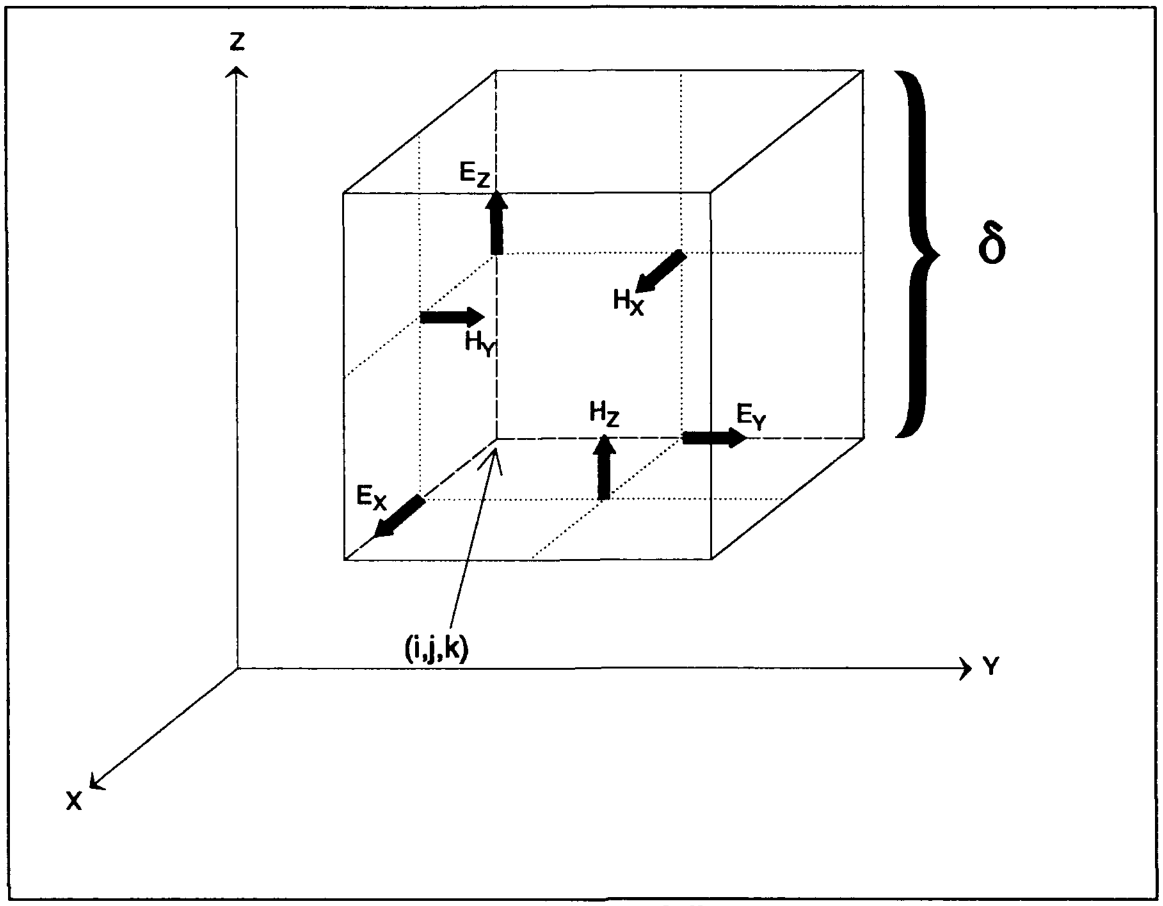

Figure 1 -- Yee Cell

The FDTD method uses centered differences which are based on the following first-order approximations to the derivative (12):

$$\frac{\partial F(i,j,k,t)}{\partial x} = \frac{F(i+\frac{\delta x}{2},j,k,t) - F(i-\frac{\delta x}{2},j,k,t)}{\delta x} + O(\delta x^2) \tag{9}$$

$$\frac{\partial F(i,j,k,t)}{\partial t} = \frac{F(i,j,k,t+\frac{\Delta t}{2}) - F(i,j,k,t-\frac{\Delta t}{2})}{\Delta t} + O(\Delta t^2) \tag{10}$$

The derivatives in space and time in Maxwell's equations are replaced by these centered differences. Evaluation of the values of *E* and *H* fields are offset in space by one half intervals as shown in Figure 1 (2:303). Notice that the *H* field values are defined as entering the cell and the E field values are defined along the three orthogonal edges nearest to the origin (indexes i,j,k are positive valued) and, in this study,

$$\delta = \delta x = \delta y = \delta z \tag{11}$$

E and H are also offset in time by one half intervals. The FDTD method solves alternately for E and H as time is incremented in one half time steps. The individual equations are as follows:

$$\begin{aligned} E_x^{n+1}(i+½,j,k) = {} & \frac{1 - \dfrac{\sigma_e(i+½,j,k)}{2\varepsilon(i+½,j,k)}}{1 + \dfrac{\sigma_e(i+½,j,k)}{2\varepsilon(i+½,j,k)}} E_x^n(i+½,j,k) \\ & + \frac{\Delta t}{\varepsilon(i+½,j,k)\delta} * \frac{1}{1 + \dfrac{\sigma_e(i+½,j,k)}{2\varepsilon(i+½,j,k)}} \\ & * \left[\begin{matrix} H_z^{n+½}(i+½,j+½,k) - H_z^{n+½}(i+½,j-½,k) \\ + H_y^{n+½}(i+½,j,k-½) - H_y^{n+½}(i+½,j,k+½) \end{matrix} \right] \end{aligned} \tag{12}$$

$$\begin{aligned} E_y^{n+1}(i,j+½,k) = {} & \frac{1 - \dfrac{\sigma_e(i,j+½,k)}{2\varepsilon(i,j+½,k)}}{1 + \dfrac{\sigma_e(i,j+½,k)}{2\varepsilon(i,j+½,k)}} E_y^n(i,j+½,k) \\ & + \frac{\Delta t}{\varepsilon(i,j+½,k)\delta} * \frac{1}{1 + \dfrac{\sigma_e(i,j+½,k)}{2\varepsilon(i,j+½,k)}} \\ & * \left[\begin{matrix} H_x^{n+½}(i,j+½,k+½) - H_x^{n+½}(i,j+½,k-½) \\ + H_z^{n+½}(i-½,j+½,k) - H_z^{n+½}(i+½,j+½,k) \end{matrix} \right] \end{aligned} \tag{13}$$

$$\begin{aligned} E_z^{n+1}(i,j,k+½) = {} & \frac{1 - \dfrac{\sigma_e(i,j,k+½)}{2\varepsilon(i,j,k+½)}}{1 + \dfrac{\sigma_e(i,j,k+½)}{2\varepsilon(i,j,k+½)}} E_z^n(i,j,k+½) \\ & + \frac{\Delta t}{\varepsilon(i,j,k+½)\delta} * \frac{1}{1 + \dfrac{\sigma_e(i,j,k+½)}{2\varepsilon(i,j,k+½)}} \\ & * \left[\begin{matrix} H_y^{n+½}(i+½,j,k+½) - H_y^{n+½}(i-½,j,k+½) \\ + H_x^{n+½}(i,j-½,k+½) - H_x^{n+½}(i,j+½,k+½) \end{matrix} \right] \end{aligned} \tag{14}$$

$$H_x^{n+½}(i,j+½,k+½)=\frac{1-\dfrac{\sigma_m(i,j+½,k+½)\Delta t}{2\mu(i,j+½,k+½)}}{1+\dfrac{\sigma_m(i,j+½,k+½)\Delta t}{2\mu(i,j+½,k+½)}}H_x^{n-½}(i,j+½,k+½)$$
$$+\frac{\Delta t}{\mu(i,j+½,k+½)\delta}*\frac{1}{1+\dfrac{\sigma_m(i,j+½,k+½)\Delta t}{2\mu(i,j+½,k+½)}} \tag{15}$$
$$*\begin{bmatrix} E_y^n(i,j+½,k+1)-E_y^n(i,j+½,k) \\ +E_z^n(i,j,k+½) \quad -E_z^n(i,j+1,k+½) \end{bmatrix}$$

$$H_y^{n+½}(i+½,j,k+½)=\frac{1-\dfrac{\sigma_m(i+½,j,k+½)\Delta t}{2\mu(i+½,j,k+½)}}{1+\dfrac{\sigma_m(i+½,j,k+½)\Delta t}{2\mu(i+½,j,k+½)}}H_y^{n-½}(i+½,j,k+½)$$
$$+\frac{\Delta t}{\mu(i+½,j,k+½)\delta}*\frac{1}{1+\dfrac{\sigma_m(i+½,j,k+½)\Delta t}{2\mu(i+½,j,k+½)}} \tag{16}$$
$$*\begin{bmatrix} E_z^n(i+1,j,k+½)-E_z^n(i,j,k+½) \\ +E_x^n(i+½,j,k) \quad -E_x^n(i+½,j,k+1) \end{bmatrix}$$

$$H_z^{n+½}(i+½,j+½,k)=\frac{1-\dfrac{\sigma_m(i+½,j+½,k)}{2\mu(i+½,j+½,k)}}{1+\dfrac{\sigma_m(i+½,j+½,k)}{2\mu(i+½,j+½,k)}}H_z^{n-½}(i+½,j+½,k)$$
$$+\frac{\Delta t}{\mu(i+½,j+½,k)\delta}*\frac{1}{1+\dfrac{\sigma_m(i+½,j+½,k)}{2\mu(i+½,j+½,k)}} \tag{17}$$
$$*\begin{bmatrix} E_x^n(i+½,j+1,k)-E_x^n(i+½,j,k) \\ +E_y^n(i,j+½,k) \quad -E_y^n(i+1,j+½,k) \end{bmatrix}$$

where δ is the lattice spacing increment, Δt is the time step increment (12:685). In order to guarantee stability, the choice of time step and spacing increments should satisfy the following:

$$v_{\max}\times\Delta t\le\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}+\frac{1}{\delta z^2}\right)^{-\frac{1}{2}} \tag{18}$$

or, in our case,

$$\Delta t \le \frac{\delta}{\sqrt{3}\,v_{\max}} \tag{19}$$

where $v_{\max}$ is the maximum phase velocity within the computational domain (15:625). As presented, these equations can handle isotropic, inhomogeneous, lossy magnetic and lossy dielectric materials.

Note that these equations can all be represented in the following form (see Figure 2):

$$Field_{next} = Field_{prev} * K1 + K2 * \begin{bmatrix} Dual_1 - Dual_2 \\ + Dual_3 - Dual_4 \end{bmatrix} \tag{20}$$

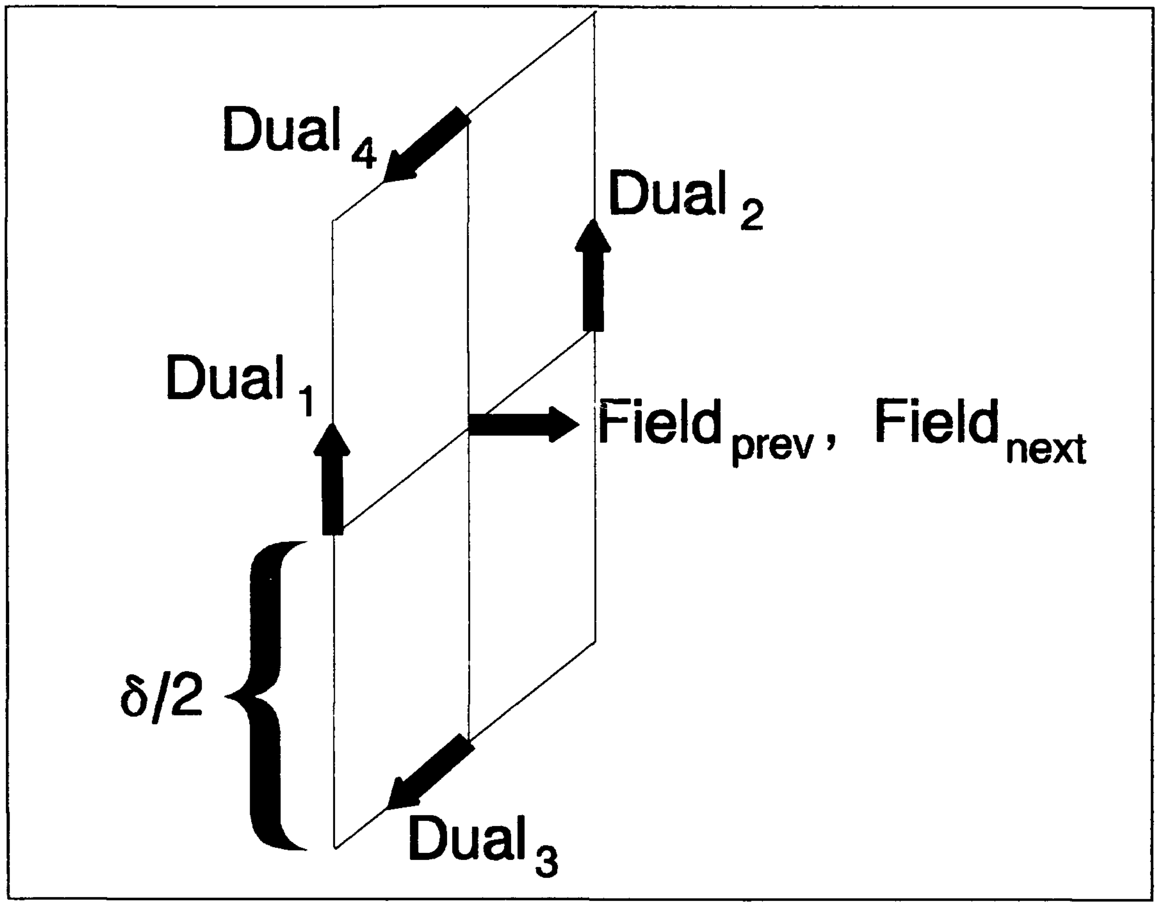

Figure 2 -- Modified Field Names

where

$$K1(i,j,k)=\frac{1-\dfrac{\sigma_e(i,j,k)}{2\varepsilon(i,j,k)}}{1+\dfrac{\sigma_e(i,j,k)}{2\varepsilon(i,j,k)}} \tag{21}$$

$$K2(i,j,k)=\frac{\Delta t}{\varepsilon(i,j,k)\delta}*\frac{1}{1+\dfrac{\sigma_e(i,j,k)}{2\varepsilon(i,j,k)}} \tag{22}$$

for equations (12)-(14) and

$$K1(i,j,k)=\frac{1-\dfrac{\sigma_m(i,j,k)}{2\mu(i,j,k)}}{1+\dfrac{\sigma_m(i,j,k)}{2\mu(i,j,k)}} \tag{23}$$

$$K2(i,j,k)=\frac{\Delta t}{\mu(i,j,k)\delta}*\frac{1}{1+\dfrac{\sigma_m(i,j,k)}{2\mu(i,j,k)}} \tag{24}$$

for equations (15)-(17), and *Dual* is the dual of the field being calculated. This simplified form leads to a straightforward method to compute these fields in hardware.

Radiation Boundary Conditions

Another computational problem area of the FDTD method is the radiation boundary condition that must be satisfied at all six faces of the volume. It arises from the fact that the fields are supposedly in an unbounded space, yet researchers lack the computational power and time to even approximate this environment. Therefore, the cell lattice is truncated along planes close to the subject of study and a radiation boundary condition is imposed. This condition attempts to determine values for the fields lying on the external boundary, since there are no fields external to these with which to calculate them using the standard cell equations. Although not nearly as computationally intense as the $O(n^3)$ FDTD cell equations problem, the calculation time for these exterior points increases as $O(n^2)$, where n is the linear dimension of the problem space. In large problems, this may account for a significant amount of time.

Many researchers using the FDTD method employ the second-order Mur radiation boundary equation, which, for the x=0 face, is (16:380):

$$
\begin{aligned}
E_z^{n+1}(0,j,k+½) = & -E_z^{n-1}(1,j,k+½) \\
& + \frac{c\Delta t-\delta}{c\Delta t+\delta} * \{E_z^{n+1}(1,j,k+½) + E_z^{n-1}(0,j,k+½)\} \\
& + \frac{2\delta}{c\Delta t+\delta} * \{E_z^{n}(0,j,k+½) + E_z^{n}(1,j,k+½)\} \\
& + \frac{(c\Delta t)^2}{2\delta(c\Delta t+\delta)} \\
& * \left[\begin{array}{lcl}
 & E_z^{n}(0,j+1,k+½) - 2*E_z^{n}(0,j,k+½) & \\
+E_z^{n}(0,j-1,k+½) & + & E_z^{n}(1,j+1,k+½) \\
-2*E_z^{n}(1,j,k+½) & + & E_z^{n}(1,j-1,k+½) \\
+E_z^{n}(0,j,k+1½) & & -2*E_z^{n}(0,j,k+½) \\
+E_z^{n}(0,j,k-½) & + & E_z^{n}(1,j,k+1½) \\
-2*E_z^{n}(1,j,k+½) & + & E_z^{n}(1,j,k-½)
\end{array} \right]
\end{aligned}
\qquad (25)
$$

A total of sixteen additions and seven multiplications are required to generate this boundary value. (The leading terms of each multiply turn out to be constant.) Combining terms to decrease the number of floating-point operations gives:

$$
\begin{aligned}
E_z^{n+1}(0,j,k+½) = & -E_z^{n-1}(1,j,k+½) \\
& + \frac{c\Delta t-\delta}{c\Delta t+\delta} * \{E_z^{n+1}(1,j,k+½) + E_z^{n-1}(0,j,k+½)\} \\
& + \frac{2\delta^2-4c\Delta t^2}{c\Delta t+\delta} * \{E_z^{n}(0,j,k+½) + E_z^{n}(1,j,k+½)\} \\
& + \frac{(c\Delta t)^2}{2\delta(c\Delta t+\delta)} \\
& * \left[\begin{array}{l}
E_z^{n}(0,j+1,k+½) + E_z^{n}(0,j-1,k+½) \\
+E_z^{n}(1,j+1,k+½) + E_z^{n}(1,j-1,k+½) \\
+E_z^{n}(0,j,k+1½) + E_z^{n}(0,j,k-½) \\
+E_z^{n}(1,j,k+1½) + E_z^{n}(1,j,k-½)
\end{array} \right]
\end{aligned}
\qquad (26)
$$

The results from this equation are based (in part) on the field values at cells to the left and right, and directly above and below the cell in question.

Simplifying further, the following expression is obtained:

$$\begin{aligned} E_z^{n+1}(0,j,k+½) = &-E_z^{n-1}(1,j,k+½) \\ &+K1*\{E_z^{n+1}(1,j,k+½)+E_z^{n-1}(0,j,k+½)\} \\ &+K2*\{E_z^{n}(0,j,k+½)+E_z^{n}(1,j,k+½)\} \\ &+K3*\begin{bmatrix} E_z^{n}(0,j+1,k+½)+E_z^{n}(0,j-1,k+½) \\ +E_z^{n}(1,j+1,k+½)+E_z^{n}(1,j-1,k+½) \\ +E_z^{n}(0,j,k+1½)+E_z^{n}(0,j,k-½) \\ +E_z^{n}(1,j,k+1½)+E_z^{n}(1,j,k-½) \end{bmatrix} \end{aligned} \tag{27}$$

where

$$K1=\frac{c\Delta t-\delta}{c\Delta t+\delta} \tag{28}$$

$$K2=\frac{2\delta^2-4c\Delta t^2}{c\Delta t+\delta} \tag{29}$$

$$K3=\frac{(c\Delta t)^2}{2\delta(c\Delta t+\delta)} \tag{30}$$

This expression now contains only twelve additions and three multiplications. It was decided that this equation could be implemented in hardware as well, so that at the conclusion of this study, the groundwork would be laid for a complete, single board FDTD computational engine capable of generating all cell and boundary field values.

Recent Advances

One of the primary limitations of FDTD is the fact that any body modeled by this method must be constructed from cubes. Even with a large number of tiny cubes, the resulting model possesses discontinuities that may not exist on the actual object, resulting in scattered fields that are not generated by a smooth surface. Recent attempts have been made to alleviate or even eliminate this requirement.

One method is to retain the cube lattice structure for the entire volume except for those cubes which intersect the surface of the object. Here the boundaries of these cubes are deformed to match that of the surface. The fields in these deformed cells are obtained by the application of Faraday's Law or Ampere's Law. These form the solutions to the fields in the area of the scatterer and are integrated into the solution for the total volume. The cells not adjoining the volume remain unchanged, enabling the use of the standard finite difference equations. Reports suggest accuracy of this method to within 1.5% of a 30-term modal solution for the scattering from a circular metal cylinder (12:688).

Another interesting method involves adapting the coordinate system to the scattering object. (This method, however, supports only two-dimensional problems. The authors report that a three-dimensional algorithm is under development.) This method surrounds the object with a curvilinear grid, which approximates a cylindrical coordinate system and closely conforms to the surface of the object. Farther away from the object, the generalized grid begins to take on the appearance of a conventional cylindrical coordinate system, until, at the outer boundary, the grid is purely cylindrical. As it turns out, the authors report no significant (order of magnitude) gains over the rectilinear method other than the fact that the radiation condition at the outer boundary is considerably simplified, since only one surface is involved in the calculation (17:88).

Researchers are also interested in the radar scattering from moving surfaces. A report by Fady Harfoush and others detail a FDTD method for determining the scattering from one and two-dimensional, perfectly conducting, relativistically moving mirrors. Apparently excellent agreement with analytical results is obtained for the case of uniform vibration and uniform translation in one dimension, and good agreement is obtained for a two-dimensional infinite vibrating mirror with oblique incidence (18:55).

Dr. Raymond Luebbers and others report on an extension of the traditional FDTD method to one capable of modelling some of the dispersive characteristics of materials. His method includes "a discrete time-domain convolution, which is efficiently evaluated using recursion." His validation of computing the wide-band reflection coefficient at an air-water boundary appears to exactly match the analytical frequency domain solution. Although the report discusses only two-dimensional problems, the report states that the extension to three dimensions is "straightforward" (19:222).

Recently, researchers have raised several issues within the context of FDTD, and as yet, these have not been resolved. Many deal with problems that arise when attempting to run simulations possessing a large dynamic range, such as computing high gain antenna patterns. Daniel Katz and others suggest the need for more study of improved boundary conditions. They also report that "the standard second-order Yee differencing algorithm may itself be unsuitable for problems" involving large dynamic ranges, saying that investigation into fourth-order methods may be necessary to reduce error (20:1210-1211).

IV. Design and Architecture of the FDTD Chip

Objective

The goal of this study was the design of a single-chip VLSI FDTD accelerator, which could be used as an individual coprocessor or as the central processing unit of a separate vector processing board in a computer (IBM/PC, workstation, Intel Hypercube). Simulations revealed the performance characteristics of this processor. Simplicity and performance were the overriding considerations for this design.

Initial Ideas

The initial idea was the development of a chip design capable of solving all six equations simultaneously (It is quite similar to the processor mentioned in reference (9), but directly attributable to a prior AFIT thesis describing a parallel approach to solving the vector wave equations (21).) Out of the desire for simplicity, it was decided to implement only one equation to illustrate the idea. If, at the conclusion of this study, more processing ability was required, then this work could be used as a first step toward the design of a simultaneous solver. (This one equation idea is also mentioned in reference (9).)

The first major decision was to work exclusively in double-precision. Although most reports today deal with single-precision, there has been some mention of making grid sizes finer, and with finer grids may come the need for double-precision. Also, as more people begin to use FDTD for precise calculations and perhaps even validation of other methods, the need for double-precision may become more apparent.

This first design would read in all seven operands (right hand side of Equation (20)) every clock cycle and also write out a result every clock cycle, once the pipeline was full. After the completion of dataflow diagrams and the writing of significant amounts of VHDL, investigations of AFIT's FPASP (Floating-Point Application Specific Processor) program suggested a second look at the design attempt.

First, the floating-point adder and multiplier on the FPASP took up about one-third of the area of the large chip (350 mil by 350 mil), while the 144 pin pads took up almost another fifth (22:6-2). The FDTD design required two multipliers and four adders, and the need for eight double-precision numbers per clock cycle would force a minimum of 512 pins. This would require a VLSI die larger than that of the FPASP and a package with twice as many pins, substantially increasing the costs of production. Second, the multiplier and adder are quite fast (25 ns cycle time, double-precision) (23). This would require large bandwidths (2.56 Gbytes/sec) to keep the chip in continuous operation. Up until this point, the assumption was that this device could be fed by a simple dynamic RAM system. Although possible, this was not practical for a simple system, since this level of bandwidth would require large bus structures or complicated interleaving strategies. Even though it was assumed from the beginning that data would be made available to the chip as fast as required, it was decided not to force the interface designer into providing these high bandwidths.

These realities brought forth the present design, a single chip containing five registers, one multiplier, and one adder, all double-precision. Data is transferred via a 64-bit data bus, with multiplexed input and output, so only 64 pins are required for data transfer. This, along with three control lines (clock, reset, and overflow), means that pin counts are relatively low. Based on the numbers in Comtois' thesis (22), chip area might be around 250 mil by 250 mil.

Numbers are read into the chip, one-at-a-time, instead of in parallel. This increases the amount of time it takes to input data, but keeps the bandwidth at about 320 Mbytes/sec. Although this bandwidth is beyond the reach of simple dynamic RAM systems, it is comfortably within the capabilities of simple, yet large and expensive, static RAM systems.

As a separate coprocessor, one would send the chip a set of data and then wait for the answer. However, since this chip could also serve as the heart of a FDTD accelerator board, great care was taken to ensure that maximum processing efficiency was obtained when the chip operated on streams of data vectors, instead of individual data elements. Even as computations are continuing on one set of data elements, new data is simultaneously being read in and being processed. Operating in this capacity, the interface logic external to the chip must be capable of generating pointers to the addresses of at least eight different locations in memory in order to access all of the operands and specify a location to store the result. Also this logic must possess some sort of counter that would signal the output of the last result and halt the FDTD chip.

Description

Figure 3 shows the overall layout of the FDTD chip design. Out of the five 64-bit registers in the design, two are the input registers to the multiplier (R1,R2), two are the input registers to the adder (R3,R4), and the last holds the computed result until it is ready for output (R5). There are six bus switches, three are one bus to two bus selectors (S1,S2,S6) and three are two bus to one bus multiplexers (S3,S4,S5). A special multiplexed bus switch (S7), connected to the input/output pins, is used to route incoming data into the chip and outgoing data from the result register. The multiplier (MUL) and adder (ADD) are practically identical to those used in the FPASP program and take two pipelined cycles to calculate results. There are a total of eighteen different buses running between the switches, registers, and floating-

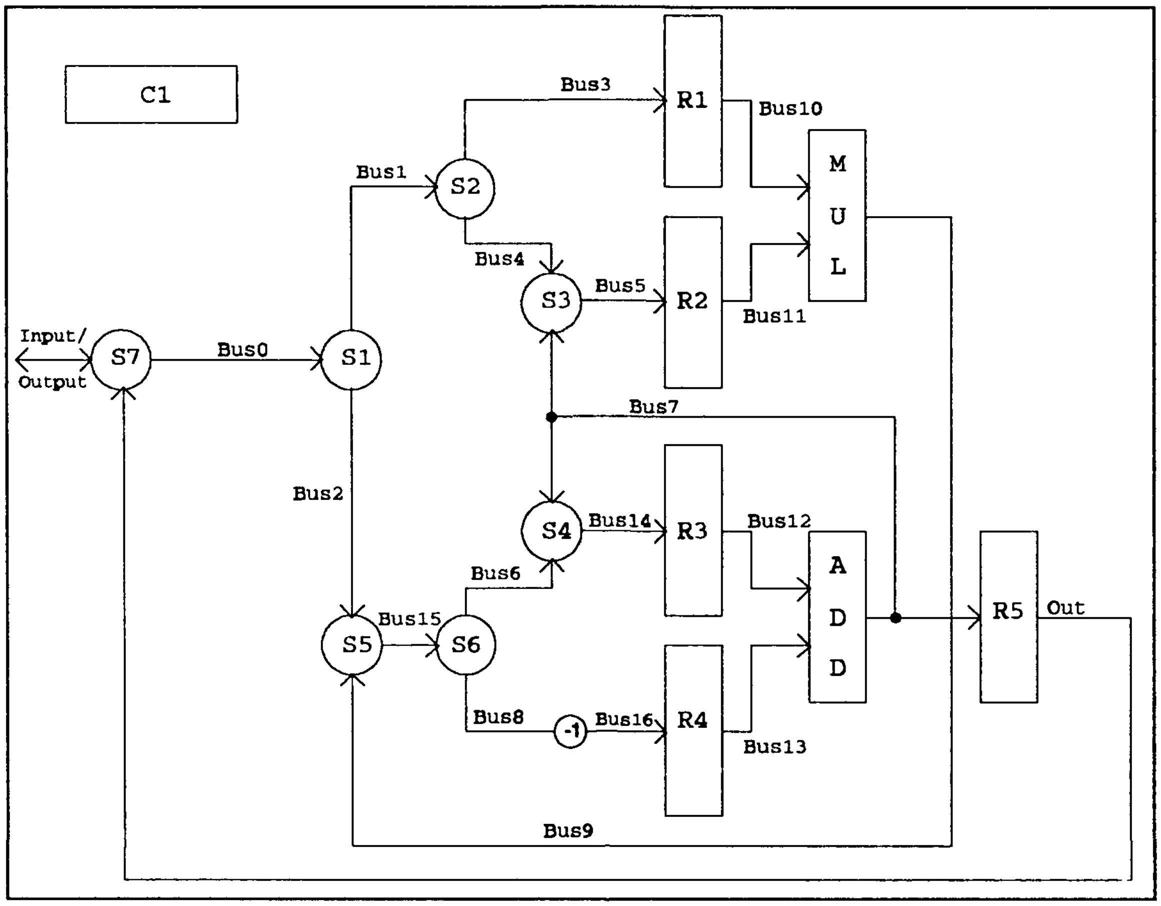

Figure 3 -- FDTD Chip Architecture

point units. An eight-state sequencer (C1) controls the operation of the above hardware, but the control signal paths are not shown in the figure to improve clarity.

The first five numbers (the first two of the four dual-field components, the previous field value, the third dual-field component, and the constant K1) are sent into the chip during cycles t_0 to t_4 (and latched at the beginning of t_1 to t_5, see Figure 4). During t_5, output data is made available to the off-chip circuitry. No data is output during the first occurrence of t_5, however, since all of the data is not yet entered and the calculations are not complete until the next occurrence of t_5. During t_6, the last dual-field component is entered and the last constant, K2, is loaded during t_7, the final tick of the cycle. A new data set is entered starting with t_0.

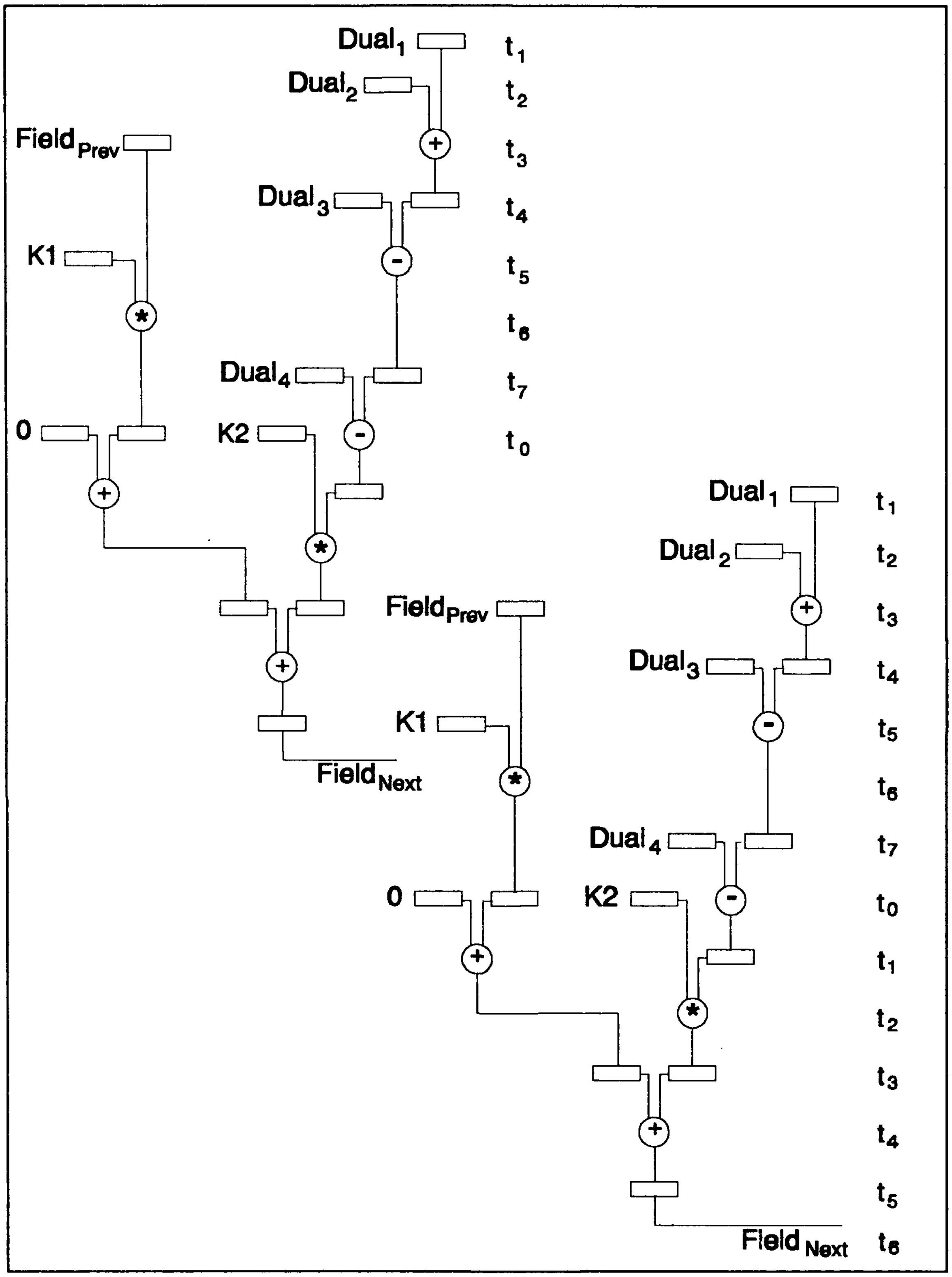

Figure 4 -- FDTD Data Sequence Diagram

During the next occurrence of t_5, the result from the previous data set is output. Therefore, the output of the result of the first data set takes 14 clock cycles, with the subsequent results following in 8 cycle intervals. (Appendix A contains a cycle-by-cycle explanation.)

VHDL Simulation

The registers' and switches' operation are modelled in VHDL behavioral descriptions (see Appendix B). The multiplier and adder consist of structural models of more basic units (which have behavioral descriptions). The VHDL code for this project is not directly compatible with the VHDL for the FPASP program, primarily since the FPASP floating-point units are driven by a two-phase non-overlapping clock, while the floating-point units in this project are driven by a single control lir.e (24). This control line signals only the onset of the second phase of the multiply (modeled as a latch of the first stage), since the first stage is considered a combinational circuit with outputs available soon after the inputs are latched. Therefore, the floating-point outputs are valid until shortly after the next operation is signaled. These differences, however, are relatively minor, and can be resolved in later stages of design. Another difference in the two VHDL representations is the fact that even for double-precision, the VHDL for the FPASP uses only single-precision calculations in modelling the behavior. The VHDL routines for the FDTD chip, on the other hand, calculate the full double-precision answer. (Even though a commitment was made to double-precision at the outset of this project, an attempt was made in the writing of the VHDL to accommodate any level of precision. This feature, however, has not been tested.)

This VHDL model is intended to not only show the behavior of this particular algorithm but also to specify an architecture that implements the behavior, in order to demonstrate feasibility and enable a study of the performance. The design put forth in this thesis, however, is not intended to be the final specific architecture; the hardware will most likely be put together from existing components and cells, and therefore minor changes to details such as rising- or falling-edge triggering, and one- or two-phase clocking are to be expected.

Items of interest include:

1. The only control signal (besides the clock) is the reset signal. Reset is asynchronous and clears all registers. Calculations begin with the first rising clock after the fall of the reset signal.

2. One goal was to keep the number of components to a minimum. During t_0 to t_3, the result of K1*Field$_{Prev}$ must be delayed until K2*(Dual-Fields) is completed. Instead of laying out another bus with a register to delay the value (and another bus switch), K1*Field$_{Prev}$ is run through the adder, with a floating-point zero as the other addend. This "null" addition effectively delays K1*Field$_{Prev}$ so that is arrives at the proper time. The floating-point zero is not actually loaded, but is created by a register reset signal.

3. The critical path involves the addition of the four dual-fields, their multiplication by K2, and finally the addition to K1*Field$_{Prev}$. Since each multiplication and addition lasts two clock cycles, the fastest time possible to achieve a result is ten clock cycles for the math operations, plus two to read in the first two operands, plus one to output the result or thirteen cycles. This FDTD chip design arrives at the first result in fourteen clock cycles. The fourteenth clock cycle provides a necessary gap in the sequence of operations to output the calculated result from the previous data set. This design, therefore, computes the first result in the nearly minimum amount of time, given the hardware available and the objective of simplicity. Note also that after the first result is output, the input/output bus is continously

operating with valid data and answers are generated every eight cycles. Based on the constraints of this study, this circuit design is highly efficient.

4. The only IEEE exception support provided is for overflow. This condition is checked in the renormalizer section of the adder, and in the exponent adder and renormalizer sections of the multiplier. This signal is made available outside the chip but has no effect on operation. It is left to the interface designer to halt the chip with the reset signal or to continue processing when an overflow exception occurs. The signal is cleared by a non-overflowing calculation.

5. Subtraction is performed by inverting the sign bit of a floating-point number and then adding. This inversion is performed by an exclusive-or gate enabled by a boolean combination of signals from the controller.

6. The input/output bus is a VHDL resolved signal type. Instead of defining a three-state logic to provide information on connections and disconnections, null assignments are used to disconnect drivers that are not permitted on the bus at that particular time. The bus resolution function therefore need only select the first element in the resolved bus array, since only one is allowed to drive the bus at one time. This simplifies the VHDL code and eliminates the need for a new type definition.

Accuracy

To best achieve a "validation" of the VHDL model, it was decided to prepare a pseudo-random input stream of double-precision real numbers, sending these (bit-vectors) to the VHDL chip description while also converting them to real number representations. Totally random numbers were avoided since these could cause overflow conditions and halt the simulation. Because VHDL currently lacks the capability to directly represent double-precision real numbers, the output of the FDTD chip was compared to results obtained from single-precision real number operations, and the relative error was quantified based on the following:

$$Rel\_Error = \frac{SP\_Field_{next} - DP\_Field_{next}}{DP\_Field_{next}} \tag{31}$$

It was initially assumed that the relative error between the two formats would be on the order of one half the least significant bit in single-precision (0.5×2^{-23} or 5.96×10^{-8}). Out of thirty-five test cases, the relative error was zero in fifteen (see Appendix C). Ten more were below 10^{-7}. However, the two largest cases were between 2×10^{-6} and 2×10^{-5}. Overall, these results show that the VHDL is operating correctly, however, the understanding of error was incorrect. It is believed that these special cases are the result of a loss of significant figures caused by subtraction of near equal numbers. To illustrate, assume that the following operations performed in both single and double-precision:

$$DP\_Field_{next} = K1 \times Field_{prev} + K2 \times \begin{bmatrix} Dual_1 - Dual_2 \\ + Dual_3 - Dual_4 \end{bmatrix} \tag{32}$$

$$SP\_Field_{next} = K1 \times Field_{prev} + K2 \times \begin{bmatrix} Dual_1 - Dual_2 \\ + Dual_3 - Dual_4 \end{bmatrix} \tag{33}$$

Consider $K1=0$, $Field_{prev}=Dual_2=Dual_4=1$ and $Dual_1=Dual_3=1+\varepsilon$. If ε is 0.5×2^{-23}, $Dual_1=Dual_3=1$ in single-precision since, in this case, ε is too insignificant to represent. Since all of the significant figures are lost, SP_Field_{next} is simply zero. However, $DP\_Field_{next}= 2\varepsilon$ which, when converted to a single-precision representation, is still 2ε. This yields a relative error of 100%, an extreme case, but it demonstrates that combinations of numbers which lose their significance during the calculations can (correctly) exhibit large relative error.

Timing

The timing simulations showed that the first output of the FDTD VHDL description occurred at 336 ns when reset fell at the leading edge of the first clock pulse, and at 360 ns when reset fell anywhere else within the first clock pulse. Subsequent outputs occur at 192 ns intervals (see Appendix C). These numbers correspond to fourteen cycles for the first output and eight cycles for every output thereafter, given a 24 ns clock. (The VHDL did not recognize the fraction portion of the 12.5 ns half cycle). Since the parts in this study are specified for operation at 40 MHz, the first output is assumed to take place at 350 ns, with subsequent outputs occurring at 200 ns intervals.

Assessing Impact

In order to characterize the impact a FDTD chip would have on electromagnetic analysis, it was necessary to obtain a working FDTD FORTRAN code and make time measurements. (All measurements were obtained on Sun SPARCstation 2 workstations.) Dr. Raymond Luebbers of Penn State had provided a FDTD code to Aeronautical Systems Division of which he allowed the use for these timing measurements (25). This code operates on a 66x66x66 cell computational domain, running 1024 time steps. It appears that this size allows the data structures to reside entirely in main memory, without resorting to memory paging to and from disk. Just the storage of the cell fields and material types alone occupies almost eight Mbytes of RAM.

First, the code was run as provided, with no changes at all. Next, the code was modified to reflect the presence of a FDTD chip that would perform the FDTD equations. Since no chip is actually available, a suitable method of simulating its presence had to be determined.

It was decided that an easy way to simulate the FDTD chip would be to replace the mathematical expressions involved with a series of assignments to specific variables. This would simulate the moving of the operands to a special location in memory (most likely a small region of 20-25 ns RAM, where the FDTD chip could operate at top speed). Also, the variable that was to receive the result (in the original code) would also be assigned some value, to simulate the transfer of the output of the FDTD chip back to regular memory. The only piece missing was the actual execution time of the FDTD chip. This was based on how many results the chip would calculate and was added back into the execution times. As it turned out, the FDTD code was well documented so these modifications proved to be an easy task.

The FDTD chip was designed for the non-dispersive, inhomogeneous, lossy magnetic and electric materials, based on the original equations. The FDTD code, however, was even more flexible, in that it could handle the above materials as well as some dispersive ones. Since the code's lossy dielectric equations were set up under a different formulation, and substituting the chip might constrain the types of problems that the code could solve, it was decided to only make comparisons using the free-space capability of both the code and the chip. Thus, the simulations with the FDTD code possessed no scatterer, just empty space. Note that this generates conservative claims, since a scatterer would have no impact on the FDTD chip calculation time, but causes the FDTD FORTRAN code to execute more complex expressions.

Results with an FDTD Coprocessor

The runs of the original code took 2 hours 17 minutes. With the changes described above the code ran 2 hours and 52 minutes. To this, one must add the calculation time of the FDTD chip. Assuming that the entire problem domain is free-space, the chip would be called 1024 x 1,609,920 or 1,648,558,080 times during the course of this problem. (For iteration information, see Appendix D, Table 2.) The chip takes 14 cycles to compute a result (we are not relying on its vector pipeline capability) at 25 ns per clock, leaving the total chip calculation time at 577 seconds or 9.6 minutes. The projected total runtime with the FDTD chip is about 3 hours and 2 minutes.

The reason for such lackluster results may lie in the fact that the SPARC2 is a fairly high-performance workstation. Perhaps the SPARC2 can perform a floating-point operation in the time it takes it to determine where an element of a multidimensional array is located in memory and fetch it. This would mean that a floating-point operation would take about as long as an assignment. Another reason may be that the memory cycle time is significantly slower than that of the SPARC2, therefore more time is tied up in fetching and writing data than in calculating additions and multiplies. The original program loop features one assignment (seven reads and one write), two multiplications, and four additions. The modified program features only eight assignments (eight reads and eight writes), but this is double the number of memory references in the original code. One speculation is that the SPARC2 possesses a write-through cache, so that all writes take place at main memory speed. Also, caches are often designed under the assumption that few memory accesses are writes, an assumption that our modified program appears to violate (26:448).

Vector Application of the FDTD Design

The results obtained so far assume that the FDTD chip is being used as a coprocessor, that is, the main processor sends data to the chip and then waits for the results to be generated. As stated earlier, the FDTD chip is most efficient when supplied with a constant stream of data, especially when obtained directly from the main memory of the host computer (eliminating intermediate transfers from main memory to the memory located on the FDTD board.) Again, since it was specified that only free-space exists in the computational domain, again only the free-space equations in the FDTD code were modified.

In order for the FDTD chip to run at maximum speed in this mode, however, the main memory must be as fast as the chip. In order to simulate this type of operation, it is assumed that sufficient amounts of 20-25 ns memory are present on the FDTD board, along with the previously mentioned interface logic. As stated earlier, 1-Mbit 20-ns RAM chips are readily available and 4-Mbit RAM chips are already appearing on the market, so this large amount of fast memory should not prove to be difficult to obtain. This memory must appear to be generic main memory to the host, so that the entire problem domain lies within this on-board memory and not external to it. This prevents the need for the transfer of data from external memory outside the board to memory on the board. Since the FDTD chip has direct access to this fast, on-board memory, it can operate without wait states.

The next step is to modify the code to act as if it only sends the location of the vector to the FDTD chip which then calculates a vector of results. This can be simulated by assignment statements specifying the locations of the vectors of data to be processed, the number of data elements in the vectors, and the location where the results are to be stored (see Appendix E for an example). (FORTRAN 77 is not well suited for the manipulation of data structures and is not able to pass pointers to arrays, so it is only possible to simulate this operation.) The vectors are assumed to run in the direction of increasing x. Therefore, in the code, "I" is set to a constant, with only "J" and "K" varying to locate a particular vector. This simulation assumes that the cell data is stored in "I" major order, so that increasing "I" by one gives the next higher memory location. Added to the time to run this simulation is the time the FDTD chip takes to calculate all of the elements of all the vectors, times the number of time steps. This total, compared to that above, would reveal the true speed-up (or slowdown) achieved as a result of the presence of the FDTD chip. Again, as before, the FDTD chip calculation times are for double-precision while the FDTD code uses only single-precision numbers.

Vector Results of the FDTD Design

Recall that the original code ran in 2 hours and 17 minutes. The vectorized code ran in 28 minutes. This time is increased by the estimated calculation time of the FDTD chip operating on the vectors. The execution time of the FDTD chip can be expressed by the following (in ns):

$$\text{FDTD Chip Run Time} = n \times 200 + 150 \tag{34}$$

where n is the number of sets of data elements to be calculated. Each full time step requires 12,545 calculations of vectors 64 elements in length and 12,416 calculations of vectors 65 elements in length (see Appendix D, Table 2). With 1024 total time steps, the full calculation time of the FDTD chip is 5.56 minutes. Therefore, the total run time of the problem with the FDTD chip is 33.6 minutes, a factor of four reduction in the original execution time. This simulation reveals the performance benefits of the vector operations of the FDTD chip. No longer must every value be passed individually to the chip. Instead, only the pointer to the vector in each data structure in question is passed to the FDTD logic. The FDTD board interface logic steps through the on-board memory, feeding data to the FDTD chip and writing the results back to memory.

In order to see the actual benefits of the FDTD chip with respect to just the calculation time of FDTD problem (and not the problem setup, initialization, and data reduction), the original code was modified to exclude all of the cell and boundary calculations. The run time for this "overhead" was 15 minutes. Subtracting this overhead from both configurations, the original possesses 122 minutes of single-precision FDTD calculations, while the FDTD chip configuration possesses 18.6 minutes of double-precision FDTD calculations, a speed-up of over a factor 6.5 in the actual FDTD algorithm calculation process.

Summary

This chapter introduced a design for a single-chip FDTD vector accelerator. This design evaluates the FDTD cell equations as a coprocessor or as a vector processor. The first result is calculated in fourteen clock cycles, with subsequent results following every eight cycles. Operating with a 40 MHz clock, this design develops a maximum of 30 double-precision MFLOPS. When modelled as a coprocessor, this design increased the execution times of a FDTD code on a SPARC2 workstation. Operating as a vector processor, this design reduced the execution time by a factor of four.

V. A Radiation Boundary Condition Evaluator Using the FPASP

Objective

The goal of this study was the design of a single-chip VLSI accelerator for evaluating the FDTD boundary values. This chip design can be used as an individual coprocessor or as the central processing unit of a separate vector processing board in a computer. Due to the complexity of the boundary value expression, an existing AFIT design was used. Simulations were performed to demonstrate the performance characteristics of this processor. Also, simulations were performed on the combination of this design and the FDTD chip design described earlier.

Initial Ideas

It was obvious that the complexity of the boundary equations would result in a fairly intricate custom chip. Having been exposed to the design of the Floating-Point Application Specific Processor (FPASP), it was decided that these equations would make a good candidate for the FPASP program. AFIT designed and specified the original FPASP, but the program has since been adopted by Rome Laboratory. The chip has been substantially changed and now stands at version 4.7, the version upon which much of this work is based.

The FPASP is, in its simplest form, a ROM-microcoded, high-speed floating-point unit, capable of performing one double-precision multiply and one double-precision add every two clock cycles. Figure 5 shows a simplified diagram of the FPASP (22). (Note all data paths are 32 bits wide.) Both of the floating-point units are pipelined so that a maximum of two double-precision floating-point results are generated every clock cycle (80 MFLOPS with a 25 ns clock). The chip possesses several 32-bit registers, including 25 general purpose double-

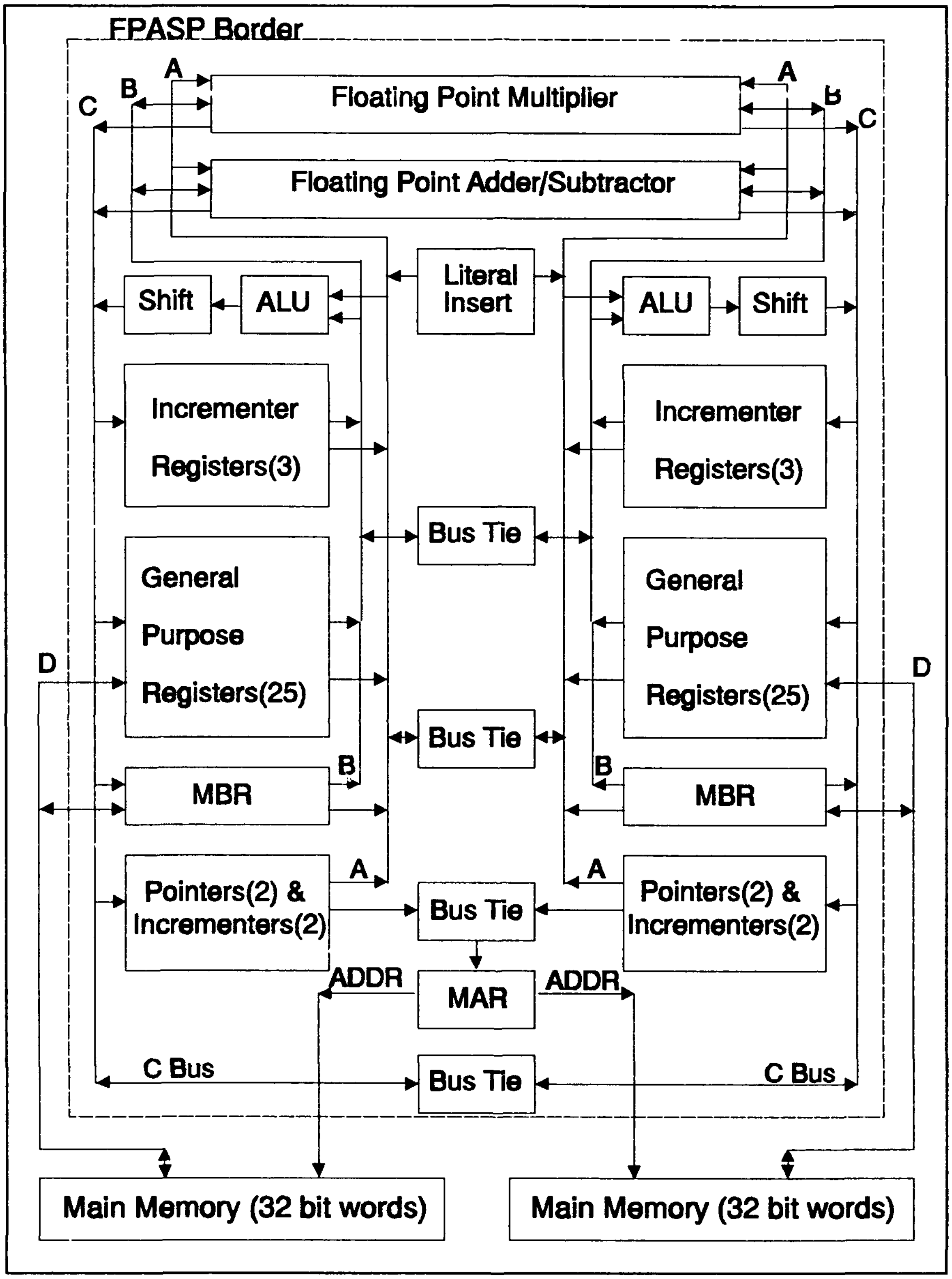
FPASP Border
A
B
C
Floating Point Multiplier
Floating Point Adder/Subtractor
Literal Insert
Shift
ALU
Incrementer Registers(3)
Bus Tie
General Purpose Registers(25)
D
MBR
Pointers(2) & Incrementers(2)
MAR
ADDR
C Bus
Main Memory (32 bit words)

Figure 5 -- FPASP Architecture

precision registers, incrementable registers, as well as memory pointer registers. In contrast to the FDTD chip described earlier, the FPASP requires little outside logic since pointers, counters, and memory signal controllers are all located on the chip. The control resides in programmable read only memory, so FPASPs can be cheaply produced in mass numbers and later microprogrammed by the individual users, each according to his needs.

Again, as with the FDTD chip, this FPASP boundary value processor was designed to operate exclusively in double-precision. And, as before, it could be used as a coprocessor to the main computer or it could be used to process vectors of data.

Plan of Attack

One of the first decisions was the order of data storage. The selection was arbitrary, since at the time this work was being performed, no actual codes had been acquired; Table 1 lists an order that was found to be useful. This order assumes calculation of z-directed E field boundary points on the x=0 face of the cube, corresponding to Equation (27). For vector operations, the data is ordered in the increasing y direction. The next task was to lay out (in time) the various operations to be performed, where operands were to be stored on the chip, and how long the floating-point units would take to compute results. At the same time, the operations had to be aligned in context with the facilities that the hardware had to offer. For example, of the three buses communicating with the floating-point units, the operands for multiplies could only come from the A and B buses. Operands for additions could come from the B and C buses, but not A. Writes going into the registers had to use the C bus. These hardware "rules" were not expressly written down, but had to be deduced from the hardware diagrams and from the list of microinstructions found in the Wafer Scale Vector Processor User's Manual (23).

Table 1 -- FPASP Memory Map of Data for Boundary Condition Evaluator

ADDRESS	UPPER MEMORY	LOWER MEMORY
0	START	ITERATIONS
.		
.		
.		
START	K1	same
+2	K2	same
+4	K3	same
+6	UP	DOWN
+8	(BLANK)	OUT
+10	$E_z^n(0,j-1,k+½)$	same
+12	$E_z^n(1,j-1,k+½)$	same
+14	$E_z^n(0,j,k+½)$	same
+16	$E_z^n(1,j,k+½)$	same
+18	$E_z^{n+1}(1,j,k+½)$	same
+20	$E_z^{n-1}(0,j,k+½)$	same
+22	$E_z^{n-1}(1,j,k+½)$	same
+24	$E_z^n(0,j+1,k+½)$	same
+26	$E_z^n(1,j+1,k+½)$	same
+28	$E_z^{n+1}(1,j+1,k+½)$	same
+30	$E_z^{n-1}(0,j+1,k+½)$	same
+32	$E_z^{n-1}(1,j+1,k+½)$	same
+34	$E_z^n(0,j+2,k+½)$	same
.		
.		
.		

Next came the writing of microcode. All important is the ordering of the fields in each line. The assembler will not recognize a field that is out of proper order. Care was taken to keep the floating-point adder as busy as possible since additions are the predominate operation in calculating the result. Multiplies were worked into time slots during and between the additions. The algorithm was condensed down to a total of twelve additions and three multiplies. This result is available after 32 clock cycles. If a vector of data is being processed, then thereafter only ten additions and three multiplies are required. These results are available every ten cycles after the first. (Note that an addition is performed every clock cycle.)

Microcode Operation

This microcode makes no assumptions on the prior state of the FPASP nor tries to preserve it. It uses the memory address register (MAR), the memory buffer register (MBR), and general purpose registers R1, R2, R4, R5, R7-R9, and R13. All register assignments (except R1 and R2) are arbitrary; other general purpose registers could easily substitute for these in an actual implementation. This algorithm uses all four variable increment registers (A INC - D INC), all four pointer registers (A PTR - D PTR), accumulators A and B (ACCA & ACCB), and the third fixed incrementer (IN3). The stack file in the floating-point unit is untouched.

The first line of microcode (for complete listing, see Appendix F) loads zero into the MAR, and sets the most significant bit of lower R1. The next line loads the address of the first 64-bit data word (constant K1) and the number of sets of data to be calculated. This statement also sets the status bits of the floating-point unit, enabling double-precision operation. The next lines read in the constants (K1, K2, and K3), pointers to previous and following rows (the k+1½ and k-½ terms or UP and DOWN), and the pointer to the location wher : the results are to be stored. These lines also set up the increments for the variable increment registers. At the ninth statement, the first of the several operands is read in, beginning the actual algorithm to calculate the boundary value. Finally, by line 32, the result of the calculations are written to memory. If more than one set of data is to be calculated, the microcode branches back to line 24 to continue calculations. Once the last result is written to memory, the program continues to line 33, which sets the DONE status bit and raises the external DONE signal.

Microcode Simulation

The microcode is compiled using the "assem" microcode compiler. "doassem" is a script file that calls "assem" as well as renames files for use by VHDL. The VHDL model of the FPASP assumes the existence of the microcode "ROM" file produced by the assembler above, as well as "RAM" files which, in this case, contain the contents of Table 1. (The actual data is in Appendix G). A mapping ROM file also provides input for the VHDL model. Since this algorithm did not use this, all elements in this ROM are set to zero. The VHDL model of the FPASP itself was created and stored in the Intermetrics work directory using the "build_inter" script file and the FPASP VHDL code, both provided by Rome Laboratory. With all of the above in place, a "sim" is performed. In our case, the VHDL FPASP completed three sets of computations in under 5 minutes on a SPARC 2 workstation (1750 ns in simulation time). During and after the simulation, several FPASP4_XXXXX.DAT files are created which chart the status of registers, buses, and so forth, as the simulation progresses. One of these files contains the upper 32 bits of RAM memory at the conclusion of the simulation. An annotated version of this file (FPASP4_U0.DAT) is in Appendix H.

It took only a few false starts to work out the errors in the microcode and generate correct results. It turned out that several unwritten rules were violated and not until these were discovered and applied did the microcode behave as expected. One is that the statement following a branch is always executed, whether the branch is taken or not. One problem (a fault of the VHDL model of the FPASP) is that the initial contents of all registers are unknown. In attempting to exclusive-or a register with itself, an unknown result was generated, when, in fact, the result is always zero, no matter what the original contents of the register. Fortunately, a zero result can be obtained by other means. If the FPASP VHDL truly modeled the hardware, this would not have been necessary.

Assessing Impact

Again, as with the FDTD chip, it was important to quantify the usefulness of an application of the FPASP in computing the boundary values. As before, the benchmarks were determined using the FDTD code supplied by Luebbers (25). The time to run the original code was compared with the time required to run a modified code which would simulate the presence of an FPASP. This was accomplished by assigning all of the operands of the boundary condition equation to new variables, which simulated the moving of these values to specific locations in memory where the FPASP could access them for its calculations. Since the program was set up for H-field radiation boundary conditions as received, only these portions of the code were modified.

FPASP Coprocessor Results

As stated above, the original code takes about 2 hours and 17 minutes to run. The modified code takes 2 hours and 17 minutes to run. The run time of the FPASP must be added to this time to get the total run time. The chip is called 23,064 x 1024 x 2 (each subroutine call performs two evaluations) or 47,235,072 times when running this problem. (For iteration information, see Appendix D, Table 2.) Since an answer is computed in 33 cycles, at 40 MHz the run time of the FPASP is about 39 seconds. The total run time with the FPASP coprocessor remains at about 2 hours and 17 minutes.

Again, it appears that finding each of the operands and performing the floating-point operations takes almost as long as finding the operands and storing them individually back to main memory. This is a disheartening result, but does not altogether signal failure, due to the method of simulating the operation of the FPASP in the code. One obvious conclusion that can be reached with these limited results is, as simulated, an FPASP chip, microprogrammed to solve the boundary condition equation, would not speed up program execution of this FDTD code as implemented on a SPARC 2 machine.

FPASP Vector Application

In order to really assess the impact of the full capability of the FPASP boundary condition solver, the FDTD code was modified to simulate the ability of the chip to process a vector of data in a single pass. In doing this two assumptions were made: one is that the passing of a pointer to a data array can be simulated with an assignment and two, that the FPASP does not require the data to be completely structured as in Table 1, but can locate all variables based on pointers and offsets. This second assumption was not accomplished in this thesis effort. However, speculation on how this might be accomplished is presented so that further studies of speed-up may be performed.

The ability of the microcode to access values to the left, right, up, and down (when working on the x=0 face) already exists. Two new pointers would be needed for the E^{n+1} and E^{n-1} values. One clock cycle would be needed before the loop begins to load these values (perhaps into the IN1 and IN2 registers). There appears to be time on the C bus in which to move these values to the MAR without adding any cycles before the loop. Inside the loop, however, the C bus is never free, so this might necessitate the addition of at most two extra statements. Note that these might be avoided by judicious use of the accumulator stacks, but this study assumes the worst case. This means the first result is available after 34 clocks and the following results are available at 12 cycle intervals. This can be represented by the following relation:

$$Boundary\ Condition\ Calc\ Time = n\times 300 + 550 \tag{35}$$

where n is the number of data sets in the vector and the time is in nanoseconds.

The FORTRAN code was modified by removing the innermost loop in the radiation subroutines. In order to access the data in the proper sequence, the following must hold: Of the three position variables I, J, and K, one is the inner loop, one is the outer, and the last takes on the values 1, 2, 3, or 4. The data must be stored in major order based on this latter variable, with the inner loop variable being the next most major. This will ensure that data is accessed correctly as the FPASP chip progresses through the vector. (This structuring was not specifically accomplished in the simulations, but could be accomplished without time penalty by re-indexing the equations which appear at the tail of each of the radiation subroutines in the FDTD code.)

FPASP Vector Results

Recall, the time to run the original FDTD code was 2 hours and 17 minutes. The time to run the modified program was 2 hours and 17 minutes. The FPASP must compute 124 vectors of length 61, 370 vectors of length 62, and 248 vectors of length 63 every time step. With 1024 time steps, the execution time of the FPASP chip was 14.6 seconds, so the total execution time of the code remained 2 hours and 17 minutes. This result was completely unexpected and does not fit with previous and following measurements. To verify the accuracy of the simulated code, selected cases were run on one node of AFIT's ORION (ELXSI) computer. As expected, the radiation boundary vectorized code displayed an acceptable amount of reduction in execution time on the ELXSI compared to the original code (537 minutes for the vectorized versus 568 minutes for the original). Since the explanation for this result on the SPARC2 is not obvious to the author and since a thorough investigation is beyond the scope of this work, no further studies were made.

Results of the Combination of FDTD and FPASP Chips

Code simulating the presence of both of the FDTD chip and the FPASP chip (a complete FDTD engine) was timed to see how it might speed up the calculations operating on vector data structures. This FORTRAN code ran in about 22 minutes. To this was added the processing times of the FDTD chip (5.6 minutes) and the FPASP chip (14.6 seconds) for a total run time of 27.8 minutes. (See Table 3, Appendix D for all execution times.) Given the original code had a run time of 137 minutes, this dual chip engine reduced run time by a factor of 4.9. In order to isolate the actual effect of the chips on just the pure FDTD calculations, the overhead was subtracted from both calculations, giving a calculation time of 122 minutes for the original FDTD code calculations, and 12.8 minutes for the calculation time for the FDTD engine. As Figure 6 shows, these numbers revealed a speed-up factor of 9.5 for just the calculation of the FDTD cell and radiation boundary condition equations alone. This was still conservative considering that problems involving magnetic or electric loss would slow down the FDTD code, but would have no effect on the calculation time of the FDTD chip. Figure 7 shows a summary of the run times discussed above. The "original" column is the run time of the FDTD code actually computing the free-space problem. The "cell" column shows the total run time using the FDTD chip design as a coprocessor for cell calculations, the "rad" column displays the total run time obtained when using the FPASP as coprocessor for boundary value calculations, and the "all" column measures the run time with both. The "vec cell" column shows the time obtained using the FDTD chip design as a vector accelerator for cell calculations, the "vec rad" displays the run time obtained using the FPASP as a vector

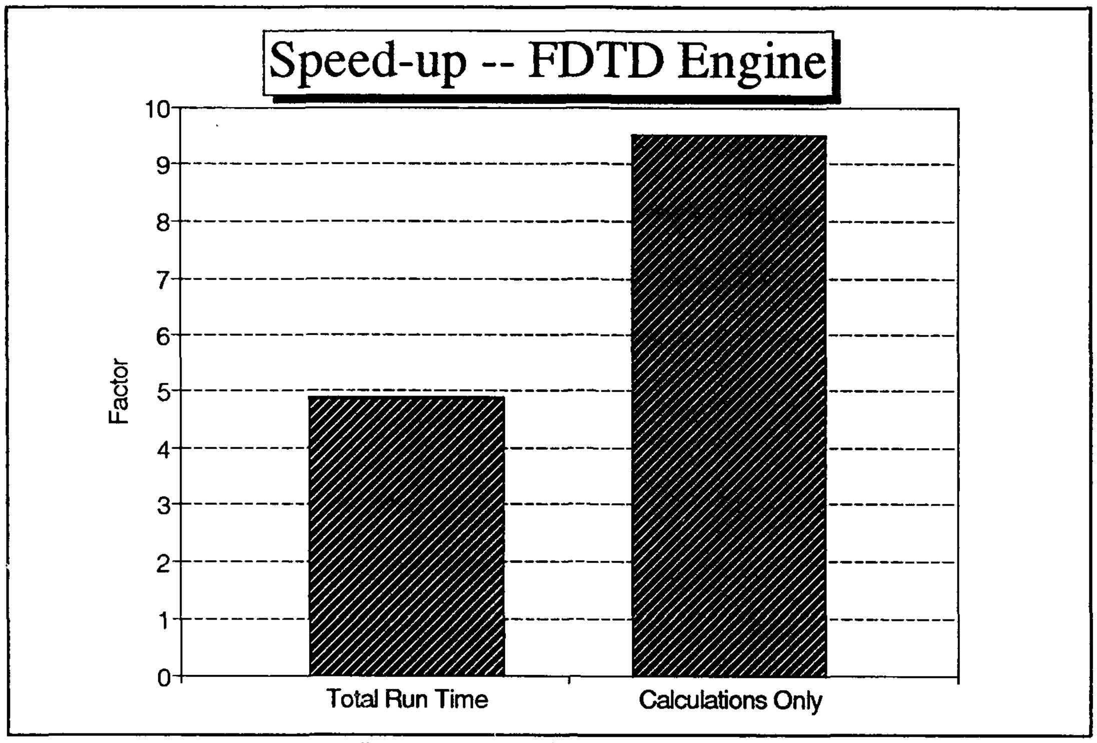

Figure 6 -- Performance Gains

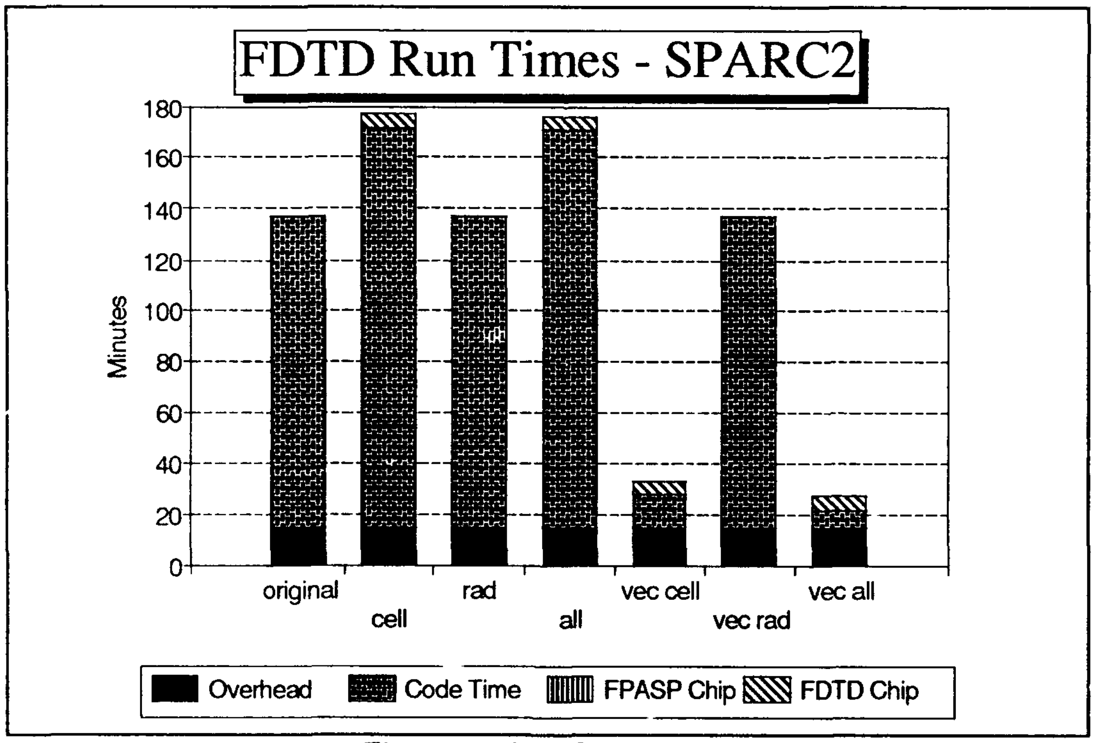

Figure 7 -- Actual Run Times

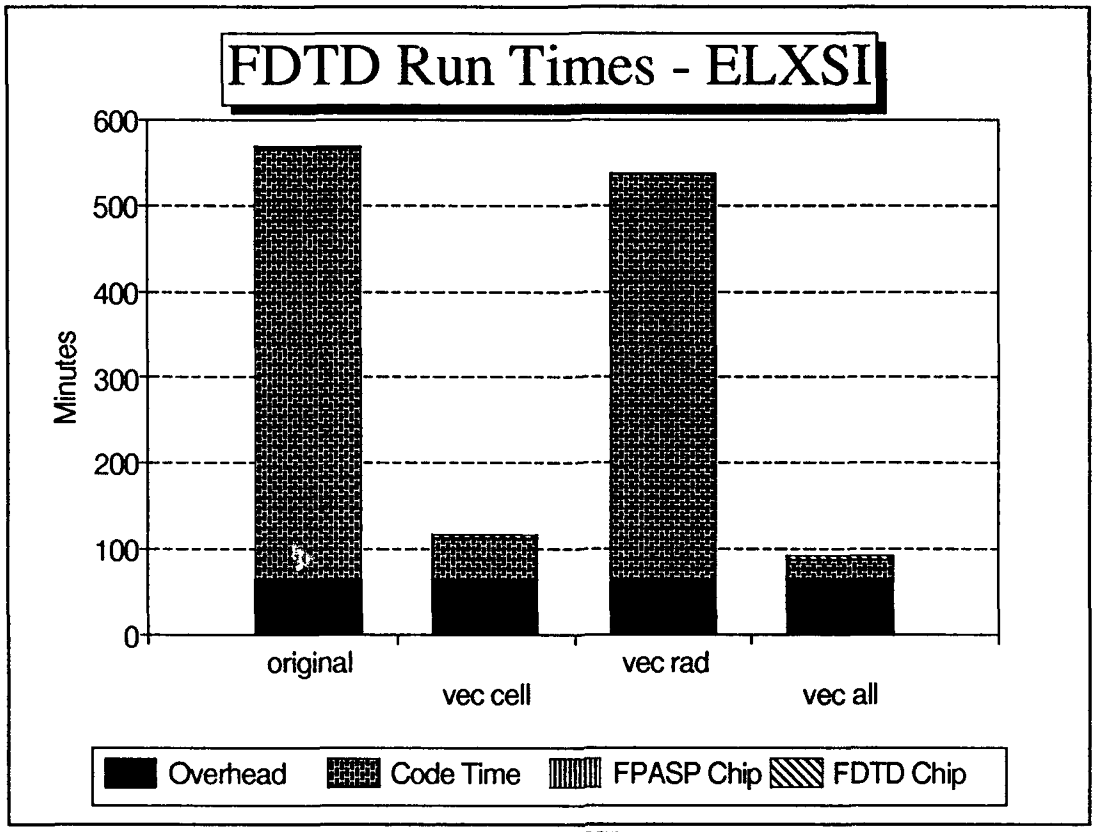

Figure 8 -- ELXSI Times

accelerator for boundary value calculations, and "vec all" show the time obtained when using both. "Code Time" is the measured execution time minus the overhead. For comparison (and to contrast the unusual time for the "vec rad" case on the SPARC2), Figure 8 shows selected run times on the ELXSI. Note that the time difference between the original and vec rad cases is the same as that between the vec cell and vec all cases.

Again, it must be emphasized that these results apply to the speed-up that one might expect on a SPARC2 workstation. For a person running FDTD on a Macintosh or IBM PC, the speed-ups would most likely be even more substantial (for example, the ELXSI times). Since the FDTD and FPASP chips would be performing the majority of the numerical operations, the total run times on these machines would compare favorably to those of the SPARC2/FDTD system as well.

Finally, even greater speed-ups would be expected if all of the vector location data for a given field component could be sent to the chip at one time. The upper limits for the speed-up in overall run time is about 9.1, while the upper limit for the speed-up in calculation time alone is about 21. These figures assume that the overhead time is the run time of the modified code (Code Time equals zero) and that the total engine computation time is still 5.8 minutes.

Summary

This chapter introduced a boundary value vector processor that is based on AFIT's FPASP design. Simulations run on VHDL descriptions of the FPASP help validate the radiation boundary condition microcode. These simulations demonstrated that as a coprocessor, the FPASP could calculate a result every 33 clock cycles at 40 MHz, for a total of about 18 MFLOPS. It is assumed that in a vector mode, the FPASP could (once the pipeline is full) generate results every 12 clock cycles, for a maximum of 43 MFLOPS. Although simulation on existing FDTD codes revealed no significant increases or decreases in run time when using the FPASP with the SPARC2, in conjunction with the previously discussed FDTD chip design, the FPASP reduced total run time from 33.6 minutes to 27.8 minutes. Together, these chips reduced code total run time by a factor of 4.9 and FDTD calculation time by a factor of 9.5.

VI. Parallel Implementations

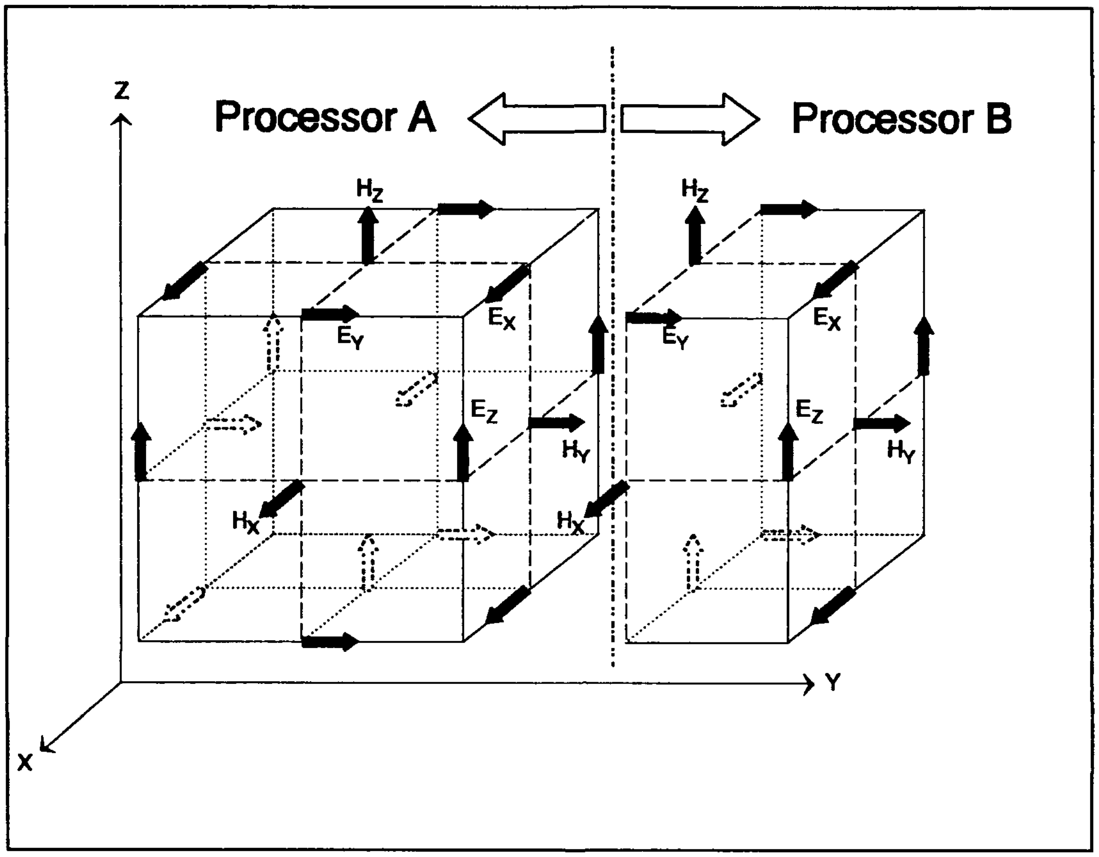

Figure 9 -- Cell Splitting Between Processors

Communication

One of the next levels of improvement to the idea of a FDTD engine is parallelization. The problem would most likely be divided up into subcubes, each assigned to an individual node. The only message passing required would be between the faces of subcubes. Assuming that E and H fields are calculated at alternate half time steps, it appears that the maximum data transfer would be two field values per cell face per half time step, or four field values per cell face per full time step. Figure 9 shows corresponding cells on either side of a split down the X-Z plane. Processor A requires H_x and H_z to calculate E_z and E_x respectively. Processor B requires E_x and E_z to calculate H_z and H_x respectively. Given large problems (while keeping the number of parallel processors constant), the computational time per node will increase on the order of n^3 (n being the length of one side of the cube), while the communication requirements will increase on the order of n^2. This suggests that the communication times on each node will be of much smaller consequence compared to the total execution time for a large problem.

Grid Scenario

A likely parallel architecture features several FDTD boards plugged into the bus of a host workstation. These would possess the same features of the board described earlier in this study as well as a DMA communications controller. This controller would be used to connect all of the board together in the form of a 2-D or 3-D grid, perhaps using optical fiber technology. The problem space would be allocated to the memory banks of each of the boards, perhaps through the system bus or through the DMA communications chip connected to dedicated disks. The geometry of the partitioning would depend on several factors, especially communication time. If communication is extremely fast, a partition favoring long vectors of data would be indicated. If communication is slow, minimizing the face area between shared processors would be necessary. (Note, however, that the communication time is not entirely additive to the previous uniprocessor case. Radiation boundary condition calculation time has been replaced by commmunication time on these shared faces.) The memory on each board might be dual partitioned such that data transfer between boards could operate on one bank, while the FDTD and FPASP operate on data in the other bank. Directions on which operations to perform would come from the host over the bus. With sixteen nodes, one could expect to solve a problem with 132 cells by 132 cells by 264 cells in about 27.8 minutes plus communication time.

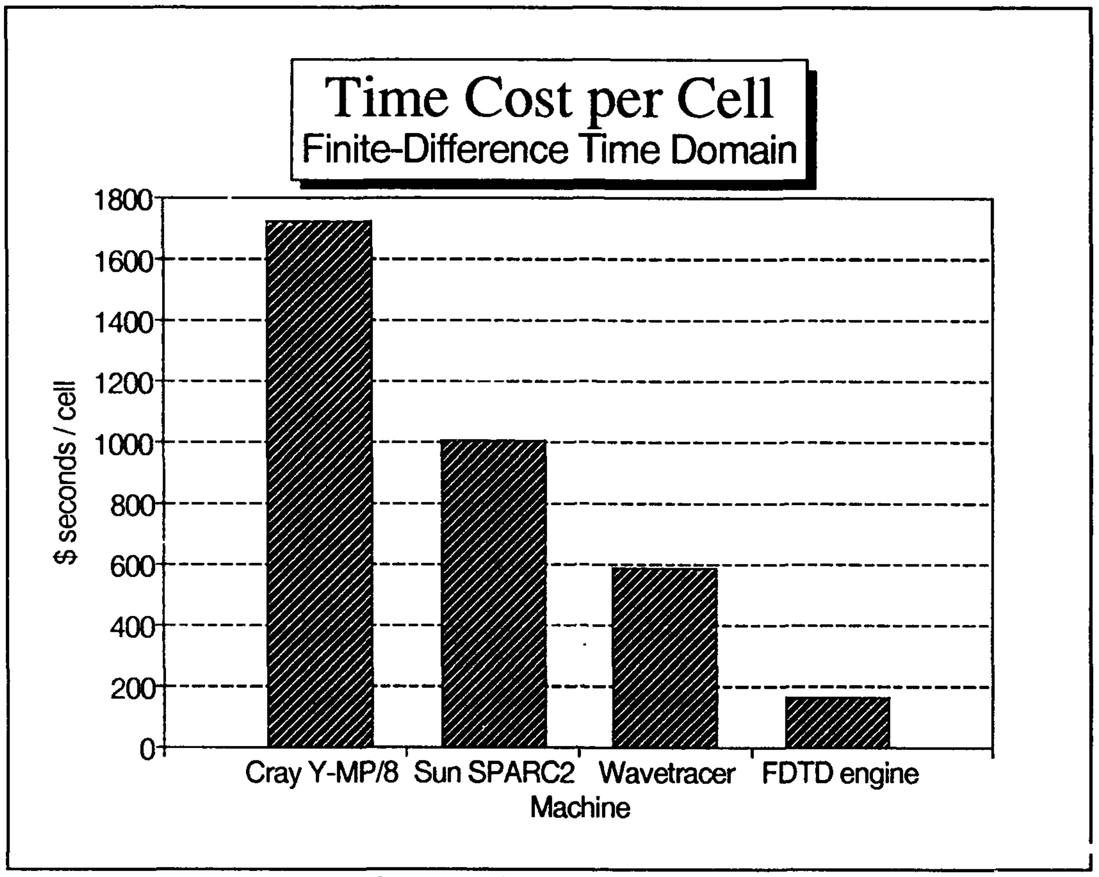

Figure 10 -- Performance Comparison of Selected Computers

For comparison, increasing the number of iterations from 1024 to 1800 (a factor of 1.76 times everything but the overhead) results in an execution time of 37.5 minutes to solve a problem with 27 million vector field unknowns. Taflove reports solving a FDTD problem with 23 million vector unknowns after 1800 time steps on a Cray Y-MP/8 in 3 minutes and 40 seconds (1). Assuming the FDTD boards could be produced for under $20,000 each, one could solve Cray size problems for $320,000, less than 10 times slower but 100 times cheaper (considering the cost of a Cray to be around $30 million (27)). Defining a performance metric as the cost of a computer times the problem run time (at 1800 time steps) divided by the number of cells in the problem (time cost per cell), the relative performance of the FDTD engine can be compared with some other computers mentioned in this study, as seen in

Figure 10 (data in Table 4, Appendix D). As one might expect, the FDTD specific computers offer lower time costs per cell than the general purpose architectures. Note that the FDTD engine is better than the Cray by a factor of 10 and the Wavetracer by a factor of 3.5. The SPARC2 is limited by the relatively small problem size that it can handle in main (RAM) memory. The usefulness of the Cray is offset by the large initial investment. The FDTD engine, on the other hand, is scalable in small dollar increments ($20,000) so that one could start small and upgrade as budget permits and problem demands. This scalability is virtually limitless since a grid communication scheme means that each node communicates with its only nearest neighbors during computations. (Of course, only a limited number of nodes could be controlled through a bus architecture, so some other means of control would be implemented for large numbers.)

Beyond the FDTD Chip

The FDTD chip, as stated earlier, performs six floating-point operations in eight 25 ns cycles, for a total maximum throughput of 30 MFLOPS. The bus interface to the chip is moving eight bytes every clock cycle (the maximum possible) for a bandwidth of 320 Mbytes per second. The FPASP chip performs thirteen floating-point operations in twelve cycles for a maximum throughput of 43 MFLOPS (out of a hardware maximum of 80 MFLOPS). The bus transfers eight bytes on nine out of every twelve cycles for an average bandwidth of 240 Mbytes per second. The figures point out the fact that even without a dedicated architecture, the FPASP is able to perform well. Indeed, because ten floating-point additions are performed every twelve cycles, even a dedicated architecture might not significantly improve the performance. It appears to be quite possible that the FPASP could perform the FDTD cell calculations just as well as the dedicated architecture described in this study.

The idea that the FPASP could perform both the cell update equations and the radiation boundary equations means that only one computational chip is needed, instead of two. Both algorithms could be microcoded in the FPASP and called by the host according to the computation to be performed at that time. Due to the ability of the FPASP to maintain and modify pointers to memory locations, much of the interface hardware required by the FDTD chip could be eliminated, simplifying board design, freeing valuable board area, and lowering costs. Given the FPASP would be produced in much larger numbers than the FDTD chip, the costs associated with a special purpose architecture would be eliminated, lowering the cost of the implementation even more, perhaps by thousands of dollars.

Beyond FDTD

Although this study has devoted itself to the development of a high-performance FDTD vector processor, one might question the rationale for spending tens of thousands of dollars for a machine that solves only electromagnetic problems. As stated above, the FPASP is a microcoded machine. Instead of microprogramming the FPASP in ROM by laser or through masking, an alternative is suggested: microprogramming the FPASP in RAM through the interface pins. This would allow the FPASP-based computer to accept a wide range of application-specific microcode, and thereby support a wide range of numerically intensive applications. Each user would practically possess the performance of a hardwired, application dedicated computer, optimized for his particular needs. Yet all users would use the same machine. This gives the machine all of the performance of a specialized computer, the flexibility and price of a workstation, and the affordable scalability of a parallel machine, allowing one to upgrade based on problem difficulty and money availability. As Figure 10 indicates, the cost premium for a general purpose computer would be eliminated, while still retaining the performance advantages of a dedicated, algorithm-specific computer.

VII. Conclusions and Recommendations

Conclusions

This study demonstrated the feasibility of a chip design which would directly and exclusively compute the FDTD cell field values. It also presented and validated an FPASP microcode program for the evaluation of the second-order Mur radiation boundary conditions. This study further demonstrated the impact of these designs on typical FDTD codes and problems. The data suggests that at least at the higher language level, these designs would have little to negative impact on run times when used as coprocessors. However, when used a vector processors, these designs appear to speed up computational time by a factor of 9.5 and corresponding total run time by a factor of 4.9. Given these designs could be fully implemented on a single board for under $20,000, one can acquire a FDTD machine running at the speed and power of ten fully parallel SPARCstation 2 workstations for the price of just two (including a $20,000 SPARC2 host). This kind of power could open the door for new research in the FDTD method and the problems it could solve, especially when this type of architecture is applied in parallel. Sixteen of these accelerator boards connected in an appropriate fashion with commercially available technology could solve problems currently being studied on eight processor Cray computers. Moreover, the time cost per cell of this performance is more than an order of magnitude less than that of a Cray, more than five times less than that of a high performance workstation such as the SPARC2, and more than three times less than that of the Wavetracer computer. The only drawback is that this accelerator, like the Wavetracer, can only solve FDTD electromagnetic problems.

In the area of architecture, this research has demonstrated the ability of the FPASP to compute algorithms other than those related to signal processing. Indeed, the FPASP was so successful at computing the complex boundary condition expressions at nearly full possible speed that it might have no trouble computing the cell field equations as well. By microprogramming the FPASP with both the cell and boundary value equations, the expenses of developing and fabricating a separate chip, as well as the costs associated with designing and producing the external interface and control circuitry required by the FDTD chip, are eliminated. Printed circuit board area would be saved, perhaps for more on-board memory or a communications controller, and the cost of the total board would be reduced. Both the cell equations and the radiation equations could be programmed onto the same chip and called by the host when required. Since the FPASP is programmed after fabrication, manufacturers could produce generic boards designed to accelerate virtually any computationally intensive application and program the ROM before shipment. This type of generic applicability means increased volume of boards resulting in lower per board costs.

Since the FPASP algorthims are microcoded there is another benefit that may be realized: reprogrammability. Once a researcher has finished studies in FDTD, he might reprogram his system to run moment method codes, finite element codes, or even fluid dynamics codes. Moreover, as new algorithms (such as improved radiation boundary conditions) become available, microcodes can be updated and the benefits immediately realized. This enables the performance benefits of algorithm dedicated hardware with the flexibility of generic computers, eliminating the only real advantage of a Cray or even a workstation. One might obtain five to ten times the computational capability for the same cost and in small increments.

Recommendations

Based on this study, the following is recommended:

1. Implement the FDTD cell equations (perhaps separating them into individual magnetic and electric, conductive and dispersive equations) and the radiation boundary condition equations (perhaps conditions more accurate than the Mur conditions used in this study) as individual, callable vector operations in microcode on an FPASP. This effort should attempt to maintain the pointers and offsets necessary to directly access all necessary operands as stored in the data structures of typical FDTD codes, so that no intermediate movement or reordering of the data is necessary. (Slight modification to the FORTRAN code may be required to accomplish this.)

2. Study, design, and perhaps fabricate a purely reprogrammable FPASP. A strong attempt should be made to preserve much of the hardware that has already been designed, since this has proven to be effective. There are two exceptions. First, the Memory Address Register (MAR) should be able to store values in the pointer registers. Second, more pointer registers would help keep track of positions in complex data structures. This is not as important as the first, but might permit more efficient microcode.

Appendix A -- FDTD Chip Timing

The movement of data into and out of the FDTD chip attempts to make maximum use of the floating point units. The calculations are pipelined in the sense that calculations on two separate results may be simultaneously proceeding at any one time. The order of operations is as follows:

t_0: The first dual-field value is made available to the chip, to be clocked in by the rising clock starting t_1.

t_1: Clocks in the first field value into register 3. The second dual-field value is made available to the chip over the input/output bus at this time.

t_2: Clocks in the second dual-field value into register 4 and marks the start of the addition operation. The previous field value is made available during this time.

t_3: Clocks in the previous field value into register 1. The third dual-field value is made available during this time. The chip inverts the sign bit of this value. The two cycle addition from the previous cycle continues.

t_4: The negated field value is loaded into register 4. The now complete sum is loaded into register 3 and a new addition begins. K1 is made available.

t_5: K1 is latched into register 2. The addition started in t_4 continues. (No value is made available since this time is reserved for output of the final result. See t_5 below.)

t_6: The final dual-field value is made available to the chip.

t_7: The multiply starts. The sum from the addition above is loaded into register 3. The final field value is clocked into register 4 and addition begins. K2 is made available to the chip.

t_0: The addition from t_7 continues. A floating point "0" is clocked into register 3 while the result of the previous multiply is clocked into register 4. The addition of these numbers begins. K2 is clocked into register 2.

t_1: The addition from t_0 continues. The result of the addition started in t_7 is clocked into register 1 and a multiply is started.

t_2: The multiply from t_1 continues.

t_3: The result from the addition are loaded into register 3 and the results from the multiplication are loaded into register 4. The addition of these numbers begins.

t_4: The addition from above continues.

t_5: The result from the addition is clocked into the output register and is put onto the input/output bus to be clocked into the off-chip circuitry.

Appendix B -- VHDL Code

This appendix lists all of the VHDL code required to simulate the FDTD chip. The bench program, however, uses the "random" procedure supplied by ZYCAD to generate random numbers. The codes are listed in alphanumerical order, according to file names. All files except SEQUENCE.VHD require the use of FDLOGIC.VHD. The VHDL hierarchy for these codes (except for FDLOGIC.VHD) is as follows:

- fdlogic.vhd
  - small.vhd
    - smallcircuit.vhd
      - 12busswitch.vhd
      - 21buswitch.vhd
      - add.vhd
        - compare.vhd
        - renormal.vhd
      - shift.vhd
      - iobusswitch.vhd
      - multiply.vhd
        - multiplier.vhd
        - renormal.vhd
      - parts.vhd
      - register.vhd
      - sequence.vhd

```
-----------------------------------------------------------------
--TITLE:                    12BUSSWITCH.VHD
--DATE:                     7 Nov 91
--VERSION:                  1.0
--PROJECT:                  Thesis
--AUTHOR:                   Raley Marek
--PROCESS:                  Switches an input bus
--                          to one of two outputs
--OPERATING SYSTEM:         UNIX
--LANGUAGE:                 VHDL
--MODULES USED:             FDLOGIC.VHD
--HISTORY/REVISIONS:        None
-----------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity busswitch_1_2 is
   port (
            Input              : in    Real_Number := blank;     -- Real_Number is collection of bits,
            Output1,Output2    : out   Real_Number := blank;     -- not a single floating point value
            control            : in    bit);                     -- '1' Output1, '0' Output2
end busswitch_1_2;

architecture behavior of busswitch_1_2 is

begin

   process (control, Input)                                      -- Combinational circuitry

   begin

      if control='1' then
         Output1 <= Input after 2 ns;
      else
         Output2 <= Input after 2 ns;
      end if;

   end process;

end behavior;
```

```
-----------------------------------------------------------------------
--TITLE:                    21BUSSWITCH.VHD
--DATE:                     7 Nov 91
--VERSION:                  1.0
--PROJECT:                  Thesis
--AUTHOR:                   Raley Marek
--PROCESS:                  Switches one of two input
--                          buses to one output bus
  OPERATING SYSTEM:         UNIX
  LANGUAGE:                 VHDL
--MODULES USED:             FDLOGIC.VHD
--HISTORY/REVISIONS:        None
-----------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity busswitch_2_1 is
   port (
          Input1, Input2   : in   Real_Number := blank;
          Output           : out  Real_Number := blank;
          control          : in   bit);              -- "1" selects Input2, "0" selects Input1
end busswitch_2_1;

architecture behavior of busswitch_2_1 is
begin
   process (control, Input1, Input2)                 -- Combinational circuit
   begin
      if control = '0' then
         Output <= Input1 after 2 ns;
      else
         Output <= Input2 after 2 ns;
      end if;
   end process;
end behavior;
```

```
-------------------------------------------------------------------------
--TITLE:                 ADD.VHD
--DATE:                  7 Nov 91
--VERSION:               1.0
--PROJECT:               Thesis
--AUTHOR:                Raley Marek
--PROCESS:               Structural description
--                       of a (2 cycle minimum)
--                       floating point adder
--OPERATING SYSTEM:      UNIX
--LANGUAGE:              VHDL
--MODULES USED:          FDLOGIC.VHD, PARTS.VHD
--HISTORY/REVISIONS:     None
-------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;
use work.Components.all;

entity Add is
   port(
      Input1, Input2   : in  Real_Number := blank;    -- Bit representation of real number
      Output           : out Real_Number := blank;    -- Same
      enable           : in  Bit;                     -- "1" clocks register for second cycle of calculations
      Over             : out Bit);                    -- Overflow signal (active high)
end Add;

architecture complete of Add is

   signal Lead1, Lead2            : Bit_Vector (0 to 0)       := "0";            -- Bit in front of binary point
   signal Zero,Pick               : Bit                       := '0';
   signal Delay, Un_Normal        : Real_Number               := blank;
   signal Shift                   : Shift_bus                 := (others=>'0');
   signal Late_Bits,Extra_Bits    : Bit_Vector (1 downto 0)   := "00";

   for all : Comparator       use entity work.Comparator       (behavior);
   for all : Shift_and_Add    use entity work.Shift_and_Add    (behavior);
   for all : FD_Register      use entity work.FD_Register      (behavior);
   for all : Renormalizer     use entity work.Renormalizer     (behavior);

begin

   C1:Comparator       port map ( Input1.exp,   Input2.exp,   Delay.exp,     Shift,
                                  Pick);

   S1:Shift_and_add    port map ( Shift,        Input1.sign,  Input2.sign,   Delay.sign,
                                  Input1.man,   Input2.man,   Delay.man,     Pick,
                                  Lead1,        Lead2,        Extra_Bits);

   F1:FD_Register      port map ( Delay,        Un_Normal,    Zero,          enable);

   R1:Renormalizer     port map ( Un_Normal,    Output,       Late_Bits,     Over);

   Lead1      <=    "0"            when Input1.exp = blank.exp        -- Lead bit 0 when number is 0
              else
                    "1";

   Lead2      <=    "0"            when Input2.exp = blank.exp        -- Same
              else
                    "1";

   Late_Bits  <=    Extra_Bits     when enable='1' and enable'activ   -- Used as register to delay a cycle
                 else
                    Late_Bits;

end complete;
```

```
--------------------------------------------------------------------------------
--TITLE:                  ADDER.VHD
--DATE:                   7 Nov 91
--VERSION:                1.0
--PROJECT:                Thesis
--AUTHOR:                 Raley Marek
--PROCESS:                Adds exponents and
--                        adjusts for offsets
--OPERATING SYSTEM:       UNIX
--LANGUAGE:               VHDL
--MODULES USED:           FDLOGIC.VHD
--HISTORY/REVISIONS:      None
--------------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity Adder is
   port (
         E1 : in      Exp_bus := (others => '0');
         E2 : in      Exp_bus := (others => '0');
         E3 : out     Exp_bus := (others => '0'));
end Adder;


architecture behavior of Adder is

begin

   process (E1,E2)

      variable adjust,holder      : Int_Array := (others => 0);
      variable temp               : Exp_bus;

   begin

      adjust(0)    := Offset;                               -- Bias determined by number of exponent bits
      holder       := Bus_to_Int(E1) + Bus_to_Int(E2);      -- Dummy variable to hold sum in array form

      if  adjust > holder then                              -- ">" defined in FDLOGIC.VHD

         E3 <= blank.exp;                                   -- Number is too small to represent

      else

         Int_to_Bus  ( holder - adjust, temp);              -- Adjust for offset and convert
         E3 <= temp after delay.exp_add;

      end if;

   end process;

end behavior;
```

```
--------------------------------------------------------------------------
--TITLE:                 COMPARE.VHD
--DATE:                  7 Nov 91
--VERSION:               1.0
--PROJECT:               Thesis
--AUTHOR:                Raley Marek
--PROCESS:               Determine the larger of two
--                       exponents for processing
--                       an addition, and determine
--                       amount of right shift for
--                       smaller number
--OPERATING SYSTEM:      UNIX
--LANGUAGE:              VHDL
--MODULES USED:          FDLOGIC.VHD
--HISTORY/REVISIONS:     None
--------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity Comparator is
    port (
        E1,E2           : in  Exp_bus   := (others => '0');
        E3              : out Exp_bus   := (others => '0');
        S1              : out Shift_bus := (others => '0');     -- How much to shift
        Select_Shift    : out bit       := '0');                -- Which one to shift
                                                                -- '1' for 1, '0' for 2
end Comparator;


architecture behavior of Comparator is

begin
   process (E1,E2)
      variable Shift_amount     : Shift_bus := (others => '0');
      variable A, B, Amount     : Int_Array := (others => 0);

   begin
      A:= Bus_to_Int(E1);
      B:= Bus_to_Int(E2);

      if B>A then
         Select_Shift  <= '1'   after delay.compare;   -- E1 is smaller
         Amount        := B - A;                       -- Subtraction only defined for positive answer
         E3            <= E2    after delay.compare;   -- Pass on the larger (E2)
      else
         Select_Shift  <= '0'   after delay.compare;   -- E2 is smaller
         Amount        := A - B;                       -- Same as above (see FDLOGIC.VHD)
         E3            <= E1    after delay.compare;   -- Pass on the larger (E1)
      end if;

      if Amount(0) > (Shift_Bus'HIGH+1)**2 then  -- If one is much greater than the other,
         Shift_amount := (others => '1');        -- all bits of the other are lost
      else
         Int_to_Bus  (Amount, Shift_amount);     -- Otherwise, determine the number of bits to shift
      end if;

      S1 <= Shift_amount;

   end process;

end behavior;
```

```
---------------------------------------------------------------------------
--TITLE:                     FDLOGIC.VHD
--DATE:                      7 Nov 91
--VERSION:                   1.0
--PROJECT:                   Thesis
--AUTHOR:                    Raley Marek
--PROCESS:                   Creates constants, types and
--                           operations for use by the
--                           FDTD chip VHDL descriptions
--OPERATING SYSTEM:          UNIX
--LANGUAGE:                  VHDL
--MODULES USED:              None
--HISTORY/REVISIONS:         None
---------------------------------------------------------------------------
package Finite_Difference_Logic is
   constant  Bits_per_Digit  : integer  := 15;                          -- # bits / integer digit
   constant  Digit_Max       : integer  := 2**Bits_per_Digit;           -- Max integer represented
   constant  Man_Bits        : integer  := 52;                          -- Mantissa ** User set  <= ?
   constant  Man_Digits      : integer  := Man_Bits/Bits_per_Digit+1; -- # of integers to hold mantissa
   constant  Exp_Bits        : integer  := 11;                          -- Exponent ** User set  <= 16?
   constant  Offset          : integer  := 2**(Exp_Bits-1)-1;           -- Exponent bias
   subtype   Man_Bus         is Bit_Vector ( Man_Bits-1 downto 0);      -- Mantissa bus
   constant  Zeros           : Man_bus  := (others => '0');             -- Empty Mantissa bus
   subtype   Shift_Bus       is bit_Vector (6 downto 0);                -- Bus to transmit amount of shift
   subtype   Exp_Bus         is Bit_Vector (Exp_Bits-1 downto 0);      -- Exponent bus

   type      Real_Number     is record                                  -- Entire digital number
      Sign      : Bit;
      Exp       : Exp_Bus;
      Man       : Man_Bus;
   end record;

   constant  Blank           : Real_Number := ('0',(others=> '0'),Zeros); -- a real number zero

   type      delays is record                                           -- Propagation delays
      exp_add    :time;
      man_add    :time;
      sign_add   :time;
      multiply   :time;
      sign       :time;
      renormal   :time;
      compare    :time;
      end record;

   constant  delay           : delays   := (0 ns, 00 ns, 0 ns, 0 ns, 0 ns, 0 ns, 0 ns);  -- *** User set
   type      Int_Array       is array (2*Man_Digits-1 downto 0)  of Integer; -- each >1 outside routines
   type      Input_Array     is array (0 to 7)                   of Real_Number;
   type      Input_Vector    is array (0 to 7)                   of Real;
   type      Real_Array      is array (Natural range<>)          of Real_Number;
   function  Real_Resolve  (Input:Real_Array) return Real_Number;  -- For in/out bus of FDTD chip
   subtype   Resolved_Bus    is Real_Resolve Real_Number;        -- Type for the in/out bus
   function  Bus_to_Int    (inbus: Bit_vector)  return Int_Array;  -- Convert bit string to int array
   function  "+"           (A, B : Int_Array)   return Int_Array;  -- Add integer arrays
   function  "-"           (A, B : Int_Array)   return Int_Array;  -- Subtract integer arrays
   function  "*"           (A, B : Int_Array)   return Int_Array;  -- Multiply integer arrays
   function  ">"           (A, B : Int_Array)   return Boolean;    -- Compare integer arrays
   function  Bus_to_Real   (A: Real_Number)     return Real;       -- Convert to real single precision #
   procedure Int_to_Bus    (int : in Int_Array; outbus : inout Bit_vector);   -- Converts int array to bits
end Finite_Difference_Logic;
```

```
package body Finite_Difference_Logic is

---------------------------------------------------------------------------
--  This function resolves a Real_Number bus by
--  selecting the first signal assigned to the bus.
--  In this implementation, only one driver is
--  ever assigned to the bus (the others are
--  disconnected with a "null" assignment), and
--  therefore the desired signal is always the first
--  in the input array.
---------------------------------------------------------------------------
   function Real_Resolve (Input:Real_Array) return Real_Number is

   begin

      if Input'LOW=0 then                                    -- Make sure something is connected
         return Input(0);
      else
         return blank;
      end if;

   end Real_Resolve;

---------------------------------------------------------------------------
-- This function takes a string of bits of limited length,
-- breaks them up into groups of Bits_per_Digit,
-- converts each group into an integer, and assigns it
-- to a corresponding position in an array.
---------------------------------------------------------------------------

   function Bus_to_Int (inbus: Bit_vector) return Int_Array is
      variable Answer                      : Int_Array    := (others=>0);
      variable Digit, Position, Advance    : integer      := 0;
      variable Convert                     : Bit_Vector (inbus'HIGH downto inbus'LOW);
   begin
      Convert := inbus;                              -- Assign to work variable
      loop
         for Index in Bits_per_Digit-1 downto 0 loop  -- Index goes high to low
            Position := Advance + Index + inbus'Low;  -- Position zig zags through bit
            If Position <= inbus'HIGH then            -- Make sure doesn't go out of bounds
               Answer(Digit) := Answer(Digit) * 2;    -- Shift by two
               If Convert(Position)='1' then
                  Answer(Digit) := Answer(Digit) +1;  -- Insert a one
               end if;
            else
               next;                                  -- Too high, try next lower
            end if;
         end loop;
         Digit      := Digit +1;                      -- Next integer array element
         Advance    := Advance + Bits_per_Digit;      -- Next group of digits
         If Advance > inbus'HIGH then
            exit;                                     -- Done
         end if;
      end loop;
      return Answer;
   end Bus_to_Int;
```

```
---------------------------------------------------------------------------
-- This function overloads the "+"
-- operator for addition on integer arrays
---------------------------------------------------------------------------

    function "+" (A, B : Int_Array) return Int_Array is
        variable C : Int_array := (others => 0);
    begin
        for i in 0 to Man_Digits loop
            C(i) := A(i) + B(i) + C(i);                 -- Add plus carry from previous
            if C(i) > Digit_Max-1 then                  -- If add is too big,
                C(i) := C(i) - Digit_Max;               -- subtract off value of next digit
                C(i+1) := 1;                            -- A carry to the next higher order digit
            end if;
        end loop;
        return C;
    end "+";

---------------------------------------------------------------------------
-- This function overloads the "-"
-- operator for subtraction on integer arrays.
-- This function REQUIRES that the first operand be
-- larger than the second, since the operation must
-- generate apositive result.
---------------------------------------------------------------------------
    function "-" (A, B : Int_Array) return Int_Array is
        variable C : Int_array := (others => 0);
    begin
        for i in Man_Digits downto 0 loop
            C(i) := A(i) - B(i);                        -- Subtract corresponding digits
        end loop;
        for i in 0 to Man_Digits loop                   -- Working up the entire array
            if C(i) <0 then                             -- If a negative result was generated,
                C(i) := C(i) + Digit_Max;               -- then add the value of the next highest digit
                C(i+1) := C(i+1)-1;                     -- and borrow from it.
            end if;
        end loop;
        return C;
    end "-";

---------------------------------------------------------------------------
-- This function overloads the "*"
-- operator for multiplication on integer arrays.
-- The operands must be less than half full inte
-- arrays, since the result requires an integer ar
-- twice the operands' size.
---------------------------------------------------------------------------
    function "*" (A, B : Int_Array) return Int_Array is
        variable C          : Int_array     := (others => 0);
        variable Index      : integer       := 0;
    begin
        for i in 0 to Man_Digits-1 loop
            for j in 0 to Man_Digits-1 loop
                Index := i+j;                           -- Maintains pointer in answer array
                C(Index) := A(i) * B(j) + C(Index);     -- Generate partial product and add to answer
                -------------------------------------------------------
                -- This code determines if an element in the answer
                -- array is too large. If so, it normalizes it to
                -- a value less than Digit_max and adds the
                -- appropriate amount to the next most significant
                -- element in the array.
                -------------------------------------------------------
                if C(Index) > Digit_Max-1 then
                    C(Index+1) := integer(C(Index)/Digit_Max) + C(Index+1);
                    C(Index)   := C(Index) mod Digit_Max;
                end if;
            end loop;
        end loop;
        return C;
    end "*";
```

```
-----------------------------------------------------------------------
-- This function overloads the ">"
-- operator for comparison of integer arrays.  Note
-- that each element in the array can contain only
-- positive values.
-----------------------------------------------------------------------
  function ">" (A, B : Int_Array) return Boolean is
  begin
    for i in Man_digits downto 0 loop          -- Go down the line

      if        A(i) > B(i) then                -- A is greater
        return true;
      elsif     A(i) < B(i) then                -- B is greater
        return false;
      end if;

    end loop;
    return false;                               -- Both are equal
  end ">";

-----------------------------------------------------------------------
-- This function takes the series of bits associated with
-- a real number and converts them into a single
-- precision real number, no matter what the  precision
-- of the original string of bits.
-----------------------------------------------------------------------

  function Bus_to_Real (A: Real_Number) return Real is
    variable Result  : Real        := 1.0;               -- Start with assumed 1.0
    variable t       : Real        := 0.5;               -- First bit position worth 0.5
    variable temp    : Int_Array   :=(others=> 0);
  begin
    for i in Man_Bus'HIGH downto 0 loop

      if A.man(i) = '1' then                    -- If a one,
        Result := Result + t;                   -- add corresponding place value
      end if;

      t := t/2.0;
                                   -- Value of significance of each bit position
    end loop;
    temp := Bus_to_Int(A.exp);     -- Convert exponent into an integer
    assert temp(0) < 1148 report "big exponent" severity warning;
    if temp(0) >1148 then          -- Exponent larger than supported by single precision
      temp(0) := 1148;             -- So just make it big
    end if;
    Result := Result * 2.0**(temp(0) - Offset);         -- Multiply by appropriate power of 2
    if A.sign = '1' then                                -- Positive or negative
      Result := Result * (-1.0);
    end if;
    return Result;
  end Bus_to_Real;
```

```
-----------------------------------------------------------------------------
-- This procedure converts a positive valued integer
-- array into a string of bits.  Note that the result
-- bit_vector is passed so that the procedure can
-- determine the limits of iteration.  One should always
-- make sure that this target contains enough room,
-- or the result is meaningless.
-----------------------------------------------------------------------------
  procedure Int_to_Bus (int : in Int_Array; outbus : inout Bit_vector) is
    variable Digit       : integer    := 0;
    variable Work        : Int_Array;
    variable Nextdigit   : integer    := Bits_per_Digit-1+outbus'LOW;
  begin
    Work := int;                                          -- Assign to work variable
    for Position in outbus'LOW to outbus'HIGH loop        -- Position goes from low to high
      if Work(Digit) mod 2 = 1 then
        outbus(Position) := '1';
      else
        outbus(Position) := '0';
      end if;
      Work(Digit) := integer(Work(Digit)/2);              -- Move the next digit to the one's place
      if Position     =  Nextdigit then                   -- If at boundary,
        Nextdigit    := Nextdigit + Bits_per_Digit;       -- move boundary
        Digit        := Digit+1;                          -- and up to next element of integer array
      end if;
    end loop;
  end Int_to_Bus;

end Finite_Difference_Logic;
```

```
-------------------------------------------------------------------------------
--TITLE:                 IOBUSSWITCH.VHD
--DATE:                  7 Nov 91
--VERSION:               1.0
--PROJECT:               Thesis
--AUTHOR:                Raley Marek
--PROCESS:               Chip interface to external
--                       world.  Manages the
--                       internal drive of the
--                       multiplexed resolved bus.
--OPERATING SYSTEM:      UNIX
--LANGUAGE:              VHDL
--MODULES USED:          FDLOGIC.VHD
--HISTORY/REVISIONS:     None
-------------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity busswitch_inout is

   port (
      R1         : inout  Resolved_Bus bus  := blank;   -- Bus interface to chip
      R2         : out    Real_Number       := blank;   -- Bus for data into chip
      R3         : in     Real_Number       := blank;   -- Bus for data leaving chip
      Selector   : in     Bit               := '0');    -- Direction of data flow, "0" in, "1" out
end busswitch_inout;

architecture behavior of busswitch_inout is

begin

   process(R1,R3,Selector)

   begin

      if Selector='0' then                               -- Data is coming in
         R1<= null;                                      -- Don't drive the bus
         R2 <= R1;                                       -- Put external data into chip
      else
         R1 <= R3;                                       -- Send internal data off of chip
      end if;

   end process;

end behavior;
```

```
------------------------------------------------------------------------
--TITLE:                     MULTIPLIER.VHD
--DATE:                      7 Nov 91
--VERSION:                   1.0
--PROJECT:                   Thesis
--AUTHOR:                    Raley Marek
--PROCESS:                   Performs multiplication of two
--                           Real_Number data types
--                           including their leading bits
--OPERATING SYSTEM:          UNIX
--LANGUAGE:                  VHDL
--MODULES USED:              FDLOGIC.VHD
--HISTORY/REVISIONS:         None
------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity multiplier is
   port (
        R1      : in    Real_Number;
        R2      : in    Real_Number;
        R3      : out   Real_number;
        X1,X2   : in    Bit                    := '0';      -- Bits leading binary point
        X3      : out   Bit_Vector (1 downto 0)  := "00");  -- Same, but two possible after multiply
end Multiplier;

architecture behavior of multiplier is

   signal  A,B,C : int_array := (others=>0);

begin

   process (R1,R2,X1,X2)

      variable temp                          : Bit_Vector (2*Man_Bits+1 downto 0); -- Ans work variable
      variable top                           : integer     := temp'HIGH;           -- Work pointer
      variable adjust, holder                : Int_Array   := (others => 0);
      variable M1, M2                        : Man_Bus     := Zeros;
      variable temp_exp, zero_exp, E1, E2 : Exp_bus      := (others => '0');

   begin

      M1 := R1.man;
      E1 := R1.exp;
      M2 := R2.man;
      E2 := R2.exp;

      Int_to_Bus (Bus_to_Int(X1&M1)*Bus_to_Int(X2&M2), temp);  -- Multiply and convert to integer

      R3.man        <= temp(top-2 downto top-1-Man_Bits)  after delay.multiply; -- Mantissa bits
      X3            <= temp (top downto top-1)            after delay.multiply;  -- Leading order bits
      R3.sign       <= R1.sign xor R2.sign                after delay.sign;
      adjust(0)     := Offset;                                          -- Floating point bias
      holder        := Bus_to_Int(E1) + Bus_to_Int(E2);

      if adjust > holder  or E1=zero_exp or E2=zero_exp then     -- Multiplication of two small numbers
         R3.exp  <= blank.exp  after delay.exp_add;             -- (or 0) results in 0
         R3.man  <= Zeros        after delay.multiply;
         X3      <= "00";
      else
         Int_to_Bus ( holder - adjust, temp_exp);                -- Re-bias the exponent and send out
         R3.exp  <= temp_exp  after delay.exp_add;
      end if;

   end process;

end behavior;
```

```
------------------------------------------------------------------------
--TITLE:                  MULTIPLY.VHD
--DATE:                   7 Nov 91
--VERSION:                1.0
--PROJECT:                Thesis
--AUTHOR:                 Raley Marek
--PROCESS:                Structural description of a
--                        (2 cycle minimum)
--                        floating point multiplier
--OPERATING SYSTEM:       UNIX
--LANGUAGE:               VHDL
--MODULES USED:           FDLOGIC.VHD
--HISTORY/REVISIONS:      None
------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;
use work.Components.all;
entity Multiply is
   port (
         Input1, Input2   : in   Real_Number;
         Output           : out  Real_Number;
         enable           : in   Bit;
         Over             : out  Bit);
end Multiply;

architecture complete of Multiply is

   signal Un_Norm, Delayed       : Real_Number                := blank;
   signal Extra_Bits, Late_bits  : Bit_Vector ( 1 downto 0)   := "00";    -- Bits leading binary point
   signal Lead1, Lead2           : Bit                        := '0';     -- Same
   signal Zero                   : Bit                        := '0';

   for all:Renormalizer   use entity work.Renormalizer   (behavior);
   for all:Multiplier     use entity work.Multiplier     (behavior);
   for all:FD_Register    use entity work.FD_Register    (behavior);

begin

   M1:Multiplier
         port map ( Input1, Input2, Un_Norm, Lead1, Lead2, Extra_Bits);

   F1: FD_Register
         port map ( Un_Norm, Delayed, Zero, enable);

   R1:Renormalizer
         port map ( Delayed, Output, Late_Bits, Over);  -- Shifts mantissa & lead bits and adjusts exp

   Lead1     <=   '0'          when Input1.exp=blank.exp      -- Lead is 0 only for 0 and denormal #s
             else
                  '1';

   Lead2     <=   '0'          when Input2.exp=blank.exp      -- Same
             else
                  '1';

             ----------------------------------------------------
             -- The following acts as the register above for the
             -- extra bits to give them the same timing as the rest
             -- of the number.
             ----------------------------------------------------
   Late_Bits <=   Extra_Bits   when enable='1' and enable'ACTIVE     -- Only on rising edge
             else
                  Late_Bits;                                         -- Otherwise, don't change

end complete;
```

```
--------------------------------------------------------------------------
--TITLE:                    PARTS.VHD
--DATE:                     7 Nov 91
--VERSION:                  1.0
--PROJECT:                  Thesis
--AUTHOR:                   Raley Marek
--PROCESS:                  List of all functional units
--OPERATING SYSTEM:         UNIX
--LANGUAGE:                 VHDL
--MODULES USED:             FDLOGIC.VHD
--HISTORY/REVISIONS:        None
--------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

package Components is

   component Comparator                                -- Exponent comparison for adder
      port (   E1,E2          : in   Exp_bus;
               E3             : out  Exp_bus;
               S1             : out  Shift_bus;
               Select_Shift   : out  Bit);
   end component;

   component Multiplier                                -- Multiplier unit only
      port (   R1,R2          : in   Real_Number;
               R3             : out  Real_Number;
               X1,X2          : in   Bit;
               X3             : out  Bit_Vector(1 downto 0));
   end component;

   component Multiply                                  -- Full multiplication, w/ renormalization
      port (   Input1, Input2 : in   Real_Number;
               Output         : out  Real_Number;
               enable         : in   Bit;
               Over           : out  Bit);
   end component;

   component Add                                       -- Full addition w/ renormalization
      port (   Input1,Input2  : in   Real_number;
               Output         : out  Real_number;
               enable         : in   Bit;
               Over           : out  Bit);
   end component;

   component Shift_and_Add                             -- Addition unit only
      port (   E1             : in   Shift_bus;
               S1,S2          : in   Bit;
               S3             : out  Bit;
               M1,M2          : in   Man_Bus;
               M3             : out  Man_Bus;
               Select_Shift   : in   Bit;
               X1,X2          : in   Bit_Vector;
               X3             : out  Bit_Vector);
   end component;

   component Renormalizer                                 -- Renormalizes numbers
      port (   R1             : in   Real_Number;
               R2             : out  Real_Number;
               E1             : in   Bit_Vector;
               Overflow       : out  Bit);
   end component;

   component FD_Register                               -- Rising edge triggered register
      port (   Input          : in   Real_Number;
               Output         : out  Real_Number;
               reset,write    : in   bit);
   end component;
```

```
    component busswitch_1_2                                          -- 1 in to 2 out bus multiplexer
       port (   Input                : in   Real_Number;
                Output1, Output2     : out  Real_Number;
                control              : in   bit);
    end component;

    component busswitch_2_1                                          -- 2 in to 1 out bus decoder
       port (   Input1, Input2       : in   Real_Number;
                Output               : out  Real_Number;
                control              : in   bit);
    end component;

    component busswitch_inout                                      -- Bi-directional bus switcher
       port (   R1                   : inout   Resolved_Bus bus;
                R2                   : out  Real_Number;
                R3                   : in   Real_Number;
                Selector             : in   Bit);
    end component;

    component Seq                                                  -- 8 state sequencer
       port (   clock                : in   bit;
                reset                : in   bit;
                control              : out  Bit_Vector(0 to 8));
    end component;

end Components;
```

```
-------------------------------------------------------------------------
--TITLE:                    REGISTER.VHD
--DATE:                     7 Nov 91
--VERSION:                  1.0
--PROJECT:                  Thesis
--AUTHOR:                   Raley Marek
--PROCESS:                  Rising edge triggered register
--OPERATING SYSTEM:         UNIX
--LANGUAGE:                 VHDL
--MODULES USED:             FDLOGIC.VHD
--HISTORY/REVISIONS:        None
-------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity FD_Register is
   port (
            Input         : in Real_Number    := blank;
            Output        : out Real_Number   := blank;
            reset,write   : in bit            := '0');
end FD_Register;

architecture behavior of FD_Register is

begin

   process (reset,write)

   begin

      if reset='1' then

         Output <= blank ;       -- Clears register

      elsif write='1' then

         Output <= Input ;       -- Transfer input to output only when write rises
                                 -- or reset falls when write is high.
      end if;

   end process;

end behavior;
```

```
------------------------------------------------------------------------
--TITLE:              RENORMAL.VHD
--DATE:               7 Nov 91
--VERSION:            1.0
--PROJECT:            Thesis
--AUTHOR:             Raley Marek
--PROCESS:            Renormalizes number by
--                    finding first occurrence of
--                    a "1" in string of lead and
--                    mantissa bits, shifts it to
--                    lead the binary point, and
--                    adjusts the exponent
--                    accordingly.
--OPERATING SYSTEM:   UNIX
--LANGUAGE:           VHDL
--MODULES USED:       FDLOGIC.VHD
--HISTORY/REVISIONS:  None
------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity Renormalizer is
   port (
         R1          : in   Real_Number  :=Blank;
         R2          : out  Real_Number  := Blank;
         E1          : in   Bit_Vector;                    -- Bits leading the binary point
         Overflow    : out Bit           := '0');
end Renormalizer;


architecture behavior of Renormalizer is

begin

   process (R1,E1)            -- Purely combinational circuit
      variable temp                               : Exp_bus;
      variable work                               : Bit_Vector(Man_Bits+E1'HIGH downto 0);
      variable holder, shift, exponent, Factor    : Int_Array              :=(others => 0);
      variable point                              : integer;
      variable Round                              : Bit_Vector(Man_Bits downto 0)  := (others=>'0');

   begin
      shift(0) := Man_Bits+1;                     -- Amount of left shift. If no 1s found, shift all bits out
      Factor(0) := 1;
      work := E1&R1.man;                          -- Leading bits and mantissa
      Overflow <= '0';                            -- Turn off overflow signal

      for i in work'HIGH downto 0 loop            -- Find first "1"
         if work(i) = '1'then
            shift(0) := Man_Bits -i;
            exit;
         end if;
      end loop;

      exponent := Bus_to_Int(R1.exp);

      if exponent(0) > 2*Offset then              -- Number already overflowed
         shift(0) := -1;
      end if;

      point := Man_Bits-shift(0);                       -- Points to "1"

      if shift(0)>0 then                                -- Left shifts required

         if shift > exponent then                       -- Makes for denormalized number
            Int_to_Bus( (others =>0), temp);
         else
            Int_to_Bus (exponent-shift,temp);
         end if;

         if shift(0) > Man_bits then                    -- No "1"s found, number is zero
```

```
            R2.exp <= zeros(Exp_Bits downto 1) after delay.exp_add;
            R2.man <= zeros after delay.renormal;
         else
            R2.exp <= temp after delay.exp_add;

            if Shift(0)=Man_Bits then                   -- Only "1" is the one in front of binary point
               R2.man <= zeros after delay.renormal;
            else                                    -- Shift correct amount, update exponent, and send out
               R2.man <= work(0oint-1 downto 0)&zeros(shift(0)-1 downto 0) after delay.renormal;
            end if;

         end if;

      else                                              -- Right or no shifts needed
         shift(0) := abs(shift(0));
         holder := exponent + shift;

         if shift(0) = 0 then                           -- No shifts needed, exponent OK
            R2.man <= work(point-1 downto 0) after delay.renormal;
         else                                           -- Right shifts needed
            Round := work(point-1 downto shift(0)-1);   -- String is Man_bits+1 in length
            Int_to_Bus(Bus_to_Int(Round) + Factor, Round);    -- Add "1" to the last bit to be lost
                                                              -- to round up.
            if Round(Round'HIGH downto 1) = Zeros then        -- Round went up to the most significant
               holder := holder + Factor;                     -- bit, so just increment exponent
            end if;

            if holder(0) > 2*Offset then                -- Overflow upon renormalization
               holder(0) := 2*Offset +1;
               R2.man <= Zeros after delay.renormal;    -- Set number equal to infinity representation
               Overflow <= '1';
            else                                        -- Else output proper mantissa
               R2.man <=Round(Round'HIGH downto 1) after delay.renormal;
            end if;

         end if;

         Int_to_Bus (holder, temp);
         R2.exp <= temp after delay.exp_add;            -- Output exponent
      end if;

      R2.sign <= R1.sign after delay.sign;              -- Output sign

   end process;

end behavior;
```

```
------------------------------------------------------------------------
--TITLE:                    SEQUENCE.VHD
--DATE:                     7 Nov 91
--VERSION:                  1.0
--PROJECT:                  Thesis
--AUTHOR:                   Raley Marek
--PROCESS:                  8 state controller.  Registers 3
--                          and 4 and adder control bits
--                          are always zero on last half of
--                          clock so that successive "1"s
--                          still have rising edges.
--OPERATING SYSTEM:         UNIX
--LANGUAGE:                 VHDL
--MODULES USED:             None
--HISTORY/REVISIONS:        None
------------------------------------------------------------------------
entity Sequencer is
   port(
         clock     : in bit;
         reset     : in bit;
         control   : out Bit_Vector(0 to 8));
end Sequencer;

architecture behavior of Sequencer is

begin
   process (clock,reset)
      variable state  : integer              :=0;
      variable code   : Bit_Vector(0 to 8)   := "000000000";    -- Notice left bit is 0, right is 8!!

   begin

      if reset='1' then                        -- Resets state but not output.  Just holds.
         state := 0;
      elsif clock='1' and not clock'QUIET then          -- Clock rising edge

         case state is                         ------------------------------------------------------------
            when 0 =>                          -- Bits mapped to switches(S), registers, floating
               code  := "001100110";           -- point units(*,+), and negator (Neg) as follows:
            when 1 =>                          --
               code  := "000011010";           --        0          S1
            when 2 =>                          --
               code  := "110000101";           --        1          S2, S5, Neg
            when 3 =>                          --
               code  := "000101110";           --        2          S3, S4, S6
            when 4 =>                          --
               code  := "100001110";           --        3          R1
            when 5 =>                          --
               code  := "110010010";           --        4          R2
            when 6 =>                          --
               code  := "000000000";           --        5          R3
            when others =>                     --
               code  := "110001101";           --        6          R4
               state  := -1;                   --
         end case;                             --        7          +
                                               --
         control   <= code;                    --        8          *
         state     := state +1;                --
      end if;                                  ------------------------------------------------------------

      if clock ='0' then                       -- Registers need rising edges to clock data.  Force to zero
         control(5 to 7) <= "000";             -- on last half of cycle so successive "1"s have rising edges
      end if;

   end process;

end behavior;
```

```
-------------------------------------------------------------------------
--TITLE:              SH'FT.VHD
--DATE:               7 Nov 91
--VERSION:            1.0
--PROJECT:            Thesis
--AUTHOR:             Raley Marek
--PROCESS:            Shifts mantissas appropriate
--                    amount and adds them
--OPERATING SYSTEM:   UNIX
--LANGUAGE:           VHDL
--MODULES USED:       FDLOGIC.VHD
--HISTORY/REVISIONS:  None
-------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;

entity Shift_and_Add is
   port (
         E1            : in   Shift_bus    := (others => '0');    -- Amount to shift smaller number
         S1,S2         : in   Bit          := '0';                -- Sign bits
         S3            : out  Bit          := '0';                -- Same
         M1,M2         : in   Man_Bus;                            -- Mantissas
         M3            : out  Man_Bus;                            -- Same
         Select_Shift  : in   bit;                    -- "1" shifts 1, "0" shifts 2
         X1 X2         : in   Bit_Vector;             -- Leading bits in front of
         X3            : out  Bit_Vector);            -- binary point
end Shift_and_Add;

architecture behavior of Shift_and_Add is

begin

   process (E1,S1,S2,M1,M2,Select_Shift,X1,X2)

      variable A1,A2,amount_hold   : Int_Array := (others => 0);
      variable Shift_amount        : integer;
      variable T3                  : Bit_Vector (X1'HIGH + M1'HIGH+2 downto 0);
      variable V1,V2               : Bit_Vector (Man_Bits + X1'HIGH downto 0);

   begin

      V1            := X1&M1;                 -- Mantissas and their leading bits
      V2            := X2&M2;
      amount_hold   := Bus_to_Int(E1);
      Shift_amount  := amount_hold(0);        -- Convert to single integer

      if Select_Shift = '0' then              -- "0" means 2nd needs shifting
         A1 :=Bus_to_Int(V1);
         A2 :=Bus_to_Int(V2(V1'HIGH downto Shift_amount));
      else                                    -- "1" means 1st needs shifting
         A1 :=Bus_to_Int(V1(V1'HIGH downto Shift_amount));
         A2 :=Bus_to_Int(V2);
      end if;

      if S1=S2 then                           -- Add the numbers
         Int_to_Bus( A1+A2,T3 );
         S3 <= S1 after delay.sign_add;       -- Sign is sign of either since they are same
      elsif A1>A2 then                        -- Subtraction required - Need to find
         Int_to_Bus(A1-A2,T3);                -- correct order to generate positive int array
         S3 <= S1 after delay.sign_add;       -- Sign is sign of largest
      else
         Int_to_Bus(A2-A1,T3);
         S3 <= S2 after delay.sign_add;       -- Sign is sign of largest
      end if;

      M3 <= T3(Man_Bits-1 downto 0)                    after delay.man_add;
      X3 <= T3(Man_Bits+X1'HIGH+1 downto Man_Bits) after delay.man_add;

   end process;

end behavior;
```

```
------------------------------------------------------------------------
--TITLE:               SMALL.VHD
--DATE:                7 Nov 91
--VERSION:             1.0
--PROJECT:             Thesis
--AUTHOR:              Raley Marek
--PROCESS:             Generates pseudo-random
--                     numbers to test the FDTD
--                     chip and compares output
--                     to single precision VHDL
--                     floating point results.
--OPERATING SYSTEM:    UNIX
--LANGUAGE:            VHDL
--MODULES USED:        FDLOGIC.VHD, ZYCAD's
--                     random number generator
--HISTORY/REVISIONS:   None
------------------------------------------------------------------------
library zycad;
use ZYCAD.distributions.random;
entity test is end test;
use work.Finite_Difference_Logic.all;
architecture structural of test is

   component FD_Circuit
      port(
         Inout_bus      : inout   Resolved_Bus bus;
         clock, reset   : in      bit;
         Overflow       : out     Bit);
   end component;

   signal Databus                                          : Resolved_Bus bus   := Blank;
   signal Stop                                             : boolean            := false;
   signal rel_error, error, expected, value, temp, templ   : real               := 0.0;
   signal clk,Over                                         : bit                := '0';
   signal reset                                            : bit                := '1';

   for all:FD_Circuit use entity work.FD_Circuit(small);

begin

   Chip : FD_Circuit
      port map ( Databus, clk, reset, Over );

   stop  <= TRUE    after 4000 ns;       -- Length of simulation

   clk   <= '1'when clk'stable(20 ns)    -- Trickery to let clock change to high immediately!
         else
            not clk  after 12.5 ns;      -- Otherwise clock period of 25 ns

   reset <= '0'     after 0 ns;          -- Allows start at absolute beginning of simulation


   stopcontrol : process                 -- Stops simulation

   begin
      wait until stop= TRUE;
      assert false report "Simulation done" severity failure;
   end process stopcontrol;


   create : process(clk,reset)              -- Generates random numbers for FDTD chip and checks result
                                         -- against the single precision calculations of VHDL

      variable A          : Input_Array  := (others => blank); -- Holds data to send to chip
      variable num,k      : real         := 0.5;               -- Used for random number generation
      variable b          : Input_vector;                      -- Real number data for VHDL calculations
      variable stage      : integer      := 0;                 -- Maintains synchronization with FDTD
chip
      variable Hold       : boolean      := true;              -- Delay state to get chip output data
```

```
begin

    if reset = '1' then
        stage := 0;
    elsif clk = '1' and not clk'QUIET then

        if stage = 0 then
            Hold := true;
            Rel_Error    <= error/value;                          -- Signal assignments for monitoring
            templ        <= temp;

            for j in 0 to 6 loop

                for i in 0 to man_bits-1 loop                     -- Generate random mantissa bits
                    random (k,num);
                    if num > 0.5 then
                        A(j).man(i) := '1';
                    else
                        A(j).man(i) := '0';
                    end if;
                end  loop;

                random (k,num);

                if num>0.5 then                              -- Generate random sign bit
                    A(j).sign := '1';
                end if;

                A(j).exp                       := "01111111111";  -- Generate actual exponent <= 0
                random(k,num);
                A(j).exp(integer(4.0*num))  := '0';
            end loop;

            random(k,num);                                   -- Set one number to zero
            A(integer(6.0*num)) := blank;

            for i in 0 to 6 loop                             -- Convert to real numbers
                b(i) := bus_to_real (A(i));
            end loop;

            temp <= b(2)*b(4)+(b(0)+b(1)-b(3)-b(5))*b(6);   -- VHDL single precision answer
        end if;

        if stage = 5  and Hold then                          -- Disconnect to allow receive of output data
            Databus <= null;
            Hold := false;
        else

            if stage = 5 then                                -- Get output data from chip
                error        <= bus_to_real(Databus) - templ;
                value        <= bus_to_real(Databus);
                expected     <= templ;
            end if;

            Databus          <= A(stage);                    -- Connect & transmit (except when 5 and
            stage            := (stage+1) mod 7;             -- hold) according to state
        end if;

    end if;

end process create;

end structural;
```

```
--------------------------------------------------------------------------------
--TITLE:                  SMALLCIRCUIT.VHD
--DATE:                   7 Nov 91
--VERSION:                1.0
--PROJECT:                Thesis
--AUTHOR:                 Raley Marek
--PROCESS:                Structural description of
--                        the FDTD chip
--OPERATING SYSTEM:       UNIX
--LANGUAGE:               VHDL
--MODULES USED:           FDLOGIC.VHD, PARTS.VHD
--HISTORY/REVISIONS:      None
--------------------------------------------------------------------------------
use work.Finite_Difference_Logic.all;
use work.Components.all;

entity FD_Circuit is
   port(
         Inout_bus       : inout  Resolved_Bus bus;
         clock, reset    : in bit;
         Overflow        : out Bit);
end FD_Circuit;

architecture small of FD_Circuit is

   signal   bus0, bus1, bus2, bus3, bus4, bus5, bus6,
            bus7, bus8, bus9, bus10, bus11, bus12,
            bus13, bus14, bus15, bus16, bus17        : Real_Number        :=Blank;

   signal   Clear, Done, Over1, Over2,
            K1, K2, K3, K4, K5, K6,                  : Bit                := '0';

   signal   Control                                  : Bit_Vector(0 to 8) := "000000000";

   for all : FD_Register       use entity work.FD_Register       (behavior);
   for all : Seq               use entity work.Sequencer         (behavior);
   for all : busswitch_2_1     use entity work.busswitch_2_1     (behavior);
   for all : busswitch_1_2     use entity work.busswitch_1_2     (behavior);
   for all : busswitch_inout   use entity work.busswitch_inout   (behavior);
   for all : Add               use entity work.Add               (complete);
   for all : Multiply          use entity work.Multiply          (complete);

begin

   A1 : Add               port map ( bus12,       bus13,   bus7,    control(7),   Over1);
   M1 : Multiply          port map ( bus10,       bus11,   bus9,    control(8),   Over2);
   R1 : FD_Register       port map ( bus3,        bus10,   reset,   control(3));
   R2 : FD_Register       port map ( bus5,        bus11,   reset,   control(4));
   R3 : FD_Register       port map ( bus14,       bus12,   Clear,   control(5));  -- "Clear", not "Reset"
   R4 : FD_Register       port map ( bus16,       bus13,   reset,   control(6));
   R5 : FD_Register       port map ( bus7,        bus17,   reset,   Done);
   S1 : busswitch_1_2     port map ( bus0,        bus1,    bus2,    control(0));
   S2 : busswitch_1_2     port map ( bus1,        bus3,    bus4,    control(1));
   S3 : busswitch_2_1     port map ( bus4,        bus7,    bus5,    control(2));
   S4 : busswitch_2_1     port map ( bus7,        bus6,    bus14,   control(2));
   S5 : busswitch_2_1     port map ( bus2,        bus9,    bus15,   control(1));
   S6 : busswitch_1_2     port map ( bus15,       bus6,    bus8,    control(2));
   S7 : busswitch_inout   port map ( Inout_bus,   bus0,    bus17,   Done);

   Done        <= control(1)   and   control(0);  -- Switches S7 & R5 for output of result
   Clear       <= reset        or    control(2);                   -- Generates 0 required by calculations
   bus16.sign  <= bus8.sign    xor   not(control(0) or control(4));  -- Negates value (just sign bit change)
   bus16.man   <= bus8.man;                                          -- Pass
   bus16.exp   <= bus8.exp;                                          -- Pass
   Overflow    <= Over1        or    Over2;               -- Signal overflow form adder or multiplier

end small;
```

Appendix C -- VHDL Results

This is the ZYCAD VHDL output for the FDTD chip. "VALUE" is the value output by the FDTD chip. The relative error is displayed roughly 48 ns after the value is output. Note that the first true value occurs at 336 ns.

```
Script started on Mon Nov 18 20:04:49 1991
ares[51]: zvsim test
ZYCAD 1076 VHDL Simulator Version 2.0b

(c) Copyright 1988,1989 ZYCAD CORPORATION. All rights reserved.

This Program is an unpublished work fully protected by the United States
copyright laws and is considered a trade secret belonging to ZYCAD
CORPORATION. It is not to be divulged or used by parties who have not
received written authorization from Zycad Corporation.

# cd test
# monitor active value rel_error
# run
0 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = NaN.0)
144 NS
    M:   ACTIVE /TEST/VALUE (value = 0.0)
192 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = NaN.0)
336 NS
    M:   ACTIVE /TEST/VALUE (value = -.104292)
384 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = -0.0)
528 NS
    M:   ACTIVE /TEST/VALUE (value = 0.0100549)
576 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = 0.0)
720 NS
    M:   ACTIVE /TEST/VALUE (value = -.242096)
768 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = -0.0)
912 NS
    M:   ACTIVE /TEST/VALUE (value = 0.051225)
960 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = -7.2724e-08)
1104 NS
    M:   ACTIVE /TEST/VALUE (value = 9.25844e-05)
1152 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = 2.82914e-06)
1296 NS
    M:   ACTIVE /TEST/VALUE (value = 0.30185)
1344 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = 0.0)
1488 NS
    M:   ACTIVE /TEST/VALUE (value = 0.00237772)
1536 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = 9.79217e-08)
1680 NS
    M:   ACTIVE /TEST/VALUE (value = 0.168917)
1728 NS
    M1:   ACTIVE /TEST/REL_ERROR (value = 8.8216e-08)
```

```
1872 NS
    M:   ACTIVE /TEST/VALUE (value = -0.0253436)
1920 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 1.46991e-07)
2064 NS
    M:   ACTIVE /TEST/VALUE (value = 0.00189692)
2112 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -6.13707e-08)
2256 NS
    M:   ACTIVE /TEST/VALUE (value = 0.14736)
2304 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -1.01121e-07)
2448 NS
    M:   ACTIVE /TEST/VALUE (value = -.448674)
2496 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -0.0)
2640 NS
    M:   ACTIVE /TEST/VALUE (value = 0.0908642)
2688 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -8.19968e-08)
2832 NS
    M:   ACTIVE /TEST/VALUE (value = 3.3044e-05)
2880 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 0.0)
3024 NS
    M:   ACTIVE /TEST/VALUE (value = 0.000149153)
3072 NS
    M1.  ACTIVE /TEST/REL_ERROR (value = 1.95128e-07)
3216 NS
    M:   ACTIVE /TEST/VALUE (value = 0.0687306)
3264 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 0.0)
3408 NS
    M:   ACTIVE /TEST/VALUE (value = 0.310541)
3456 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 0.0)
3600 NS
    M:   ACTIVE /TEST/VALUE (value = -0.0589758)
3648 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 6.31665e-08)
3792 NS
    M:   ACTIVE /TEST/VALUE (value = 0.000124374)
3840 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 1.33382e-05)
3984 NS
    M:   ACTIVE /TEST/VALUE (value = -0.000907746)
4032 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -0.0)
4176 NS
    M:   ACTIVE /TEST/VALUE (value = 0.0066361)
4224 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 0.0)
4368 NS
    M:   ACTIVE /TEST/VALUE (value = -.11466)
4416 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -0.0)
4560 NS
    M:   ACTIVE /TEST/VALUE (value = 2.0128e-05)
4608 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 9.03711e-08)
4752 NS
    M:   ACTIVE /TEST/VALUE (value = 0.0360449)
4800 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = -7.2346e-07)
4944 NS
    M:   ACTIVE /TEST/VALUE (value = 0.00191726)
4992 NS
    M1:  ACTIVE /TEST/REL_ERROR (value = 0.0)
5136 NS
    M:   ACTIVE /TEST/VALUE (value = -0.0154797)
```

```
5184 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = -1.20328e-07)
5328 NS
   M:   ACTIVE /TEST/VALUE (value = -.54597)
5376 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = -0.0)
5520 NS
   M:   ACTIVE /TEST/VALUE (value = 0.0106245)
5568 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 0.0)
5712 NS
   M:   ACTIVE /TEST/VALUE (value = -3.70337e-05)
5760 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 1.96469e-07)
5904 NS
   M:   ACTIVE /TEST/VALUE (value = 0.231103)
5952 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 6.44783e-08)
6096 NS
   M:   ACTIVE /TEST/VALUE (value = 0.0052199)
6144 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 1.78418e-07)
6288 NS
   M:   ACTIVE /TEST/VALUE (value = 0.839034)
6336 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 7.10396e-08)
6480 NS
   M:   ACTIVE /TEST/VALUE (value = 0.0270926)
6528 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 1.37502e-07)
6672 NS
   M:   ACTIVE /TEST/VALUE (value = 7.72364e-08)
6720 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 0.0)
6864 NS
   M:   ACTIVE /TEST/VALUE (value = 0.137487)
6912 NS
   M1:   ACTIVE /TEST/REL_ERROR (value = 1.08382e-07)
7000 NS
Assertion FAILURE at 7000 NS in design unit STRUCTURAL from process /TEST/STOPCONTROL:
   "Simulation done"
# quit
ares[53]: exit
ares[54]:
script done on Mon Nov 18 20:08:10 1991
```

Appendix D -- Performance, Run Time and Loop Data

Table 2 -- Iteration Count

Subroutine	i	j	k
EXSFLD	1,NX1	2,NY1	2,NZ1
EYSFLD	2,NX1	1,NY1	2,NZ1
EZSFLD	2,NX1	2,NY1	1,NZ1
HXSFLD	2,NX1	1,NY1	1,NZ1
HYSFLD	1,NX1	2,NY1	1,NZ1
HZSFLD	1,NX1	1,NY1	2,NZ1
RADEZX		3,NY1-1	2,NZ1-1
RADEYX		2,NY1-1	3,NZ1-1
RADEZY	3,NX1-1		2,NZ1-1
RADEXY	2,NX1-1		3,NZ1-1
RADEXZ	2,NX1-1	3,NY1-1	
RADEYZ	3,NX1-1	2,NY1-1	
RADHXZ	3,NX1-1	3,NY-3	
RADHYX		3,NY1-1	3,NZ-3
RADHZY	3,NX-3		3,NZ1-1
RADHXY	3,NX1-1		2,NZ-2
RADHYZ	2,NX-2	3,NY1-1	
RADHZX		2,NY-2	3,NZ1-1

Note that for Table 2, all timing runs for this work were made with NX=NY=NZ=66 and with NX1=NY1=NZ1=65. Each radiation subroutine ("RADXXX") performs two boundary point evaluations.

Table 3 -- Run Times of All Codes

CODE VERSION	TIME
ORIGINAL	137
MODIFIED CELL EQUATIONS	172
MODIFIED BOUNDARY EQUATIONS	134
ALL EQUATIONS MODIFIED	171
VECTORIZED CELL EQUATIONS	28
VECTORIZED BOUNDARY EQUATIONS	58
ALL EQUATIONS VECTORIZED	22
NO CELL OR RADIATION EQUATIONS	15
ALL BUT CELL EQUATIONS	24
ALL BUT RADIATION EQUATIONS	127

Table 4 -- Cost/Performance Data

	cost	number of cells	time
Cray Y-MP/8	$30 Million	3,833,000	220 secs
Sun SPARC2	$20,000	287,500	13,767 secs
Wavetracer	$400,000	1,049,000	1,530 secs
16 node FDTD engine w/ host	$340,000	4,600,000	2,220 secs

Appendix E -- Fully Modified Code

This listing shows the differences (UNIX "diff" command) between the vectall.f file (which simulates the presence of the FDTD chip design and the FPASP boundary point evaluator, both operating as vector processors) and the original fdtdd.f file. New variables used in assignments are prefixed "FDTD_STUDY".

```
127a128
>       CALL BUILD
1194c1195
< C     Subroutine EXSFLD modified
---
> C
1198d1198
<       IFDTD_STUDY=NX1
1203c1203
< C           DO 10 I=1,NX1
---
>           DO 10 I=1,NX1
1212c1212
< C           FREE SPACE -- ************ modified here !!!
---
> C           FREE SPACE
1214,1222c1214,1215
< C110        EXS(I,J,K)=EXS(I,J,K)+(HZS(I,J,K)-HZS(I,J-1,K))*DTEDY
< C    $                      -(HYS(I,J,K)-HYS(I,J,K-1))*DTEDZ
<  110        FDTD_STUDY1=EXS(1,J,K)
<       FDTD_STUDY2=HZS(1,J,K)
<       FDTD_STUDY3=HZS(1,J-1,K)
<       FDTD_STUDY4=DTEDY
<       FDTD_STUDY5=HYS(1,J,K)
<       FDTD_STUDY6=HYS(1,J,K-1)
<       FDTD_STUDY7=DTEDZ
---
>  110        EXS(I,J,K)=EXS(I,J,K)+(HZS(I,J,K)-HZS(I,J-1,K))*DTEDY
>      $                      -(HYS(I,J,K)-HYS(I,J,K-1))*DTEDZ
1263c1256
< C     Subroutine EYSFLD modified
---
> C
1267d1259
<       IFDTD_STUDY=NX1-1
1272c1264
< C           DO 10 I=2,NX1
---
>           DO 10 I=2,NX1
1281c1273
< C           FREE SPACE -- ******************** modified here!!!
---
> C           FREE SPACE
1283,1291c1275,1276
< C 110       EYS(I,J,K)=EYS(I,J,K)+(HXS(I,J,K)-HXS(I,J,K-1))*DTEDZ
< C    $                      -(HZS(I,J,K)-HZS(I-1,J,K))*DTEDX
<  110        FDTD_STUDY1=EYS(2,J,K)
<       FDTD_STUDY2=HXS(2,J,K)
<       FDTD_STUDY3=HXS(2,J-1,K)
<       FDTD_STUDY4=DTEDZ
<       FDTD_STUDY5=HZS(2,J,K)
<       FDTD_STUDY6=HZS(2,J,K-1)
<       FDTD_STUDY7=DTEDX
```

```
---
> 110      EYS(I,J,K)=EYS(I,J,K)+(HXS(I,J,K)-HXS(I,J,K-1))*DTEDZ
>     $                         -(HZS(I,J,K)-HZS(I-1,J,K))*DTEDX
1332c1317
< C     Subroutine EZSFLD modified
---
> C
1336d1320
<   IFDTD_STUDY=NX1-1
1341c1325
< C          DO 10 I=2,NX1
---
>          DO 10 I=2,NX1
1349a1334
> C        FREE SPACE
1351,1361c1336,1337
< C        FREE SPACE -- ***************** modified here !!!
< C
< C 110      EZS(I,J,K)=EZS(I,J,K)+(HYS(I,J,K)-HYS(I-1,J,K))*DTEDX
< C      $                       -(HXS(I,J,K)-HXS(I,J-1,K))*DTEDY
<  110     FDTD_STUDY1=EZS(2,J,K)
<      FDTD_STUDY2=HYS(2,J,K)
<      FDTD_STUDY3=HYS(2,J-1,K)
<      FDTD_STUDY4=DTEDX
<      FDTD_STUDY5=HXS(2,J,K)
<      FDTD_STUDY6=HXS(2,J,K-1)
<      FDTD_STUDY7=DTEDY
---
> 110      EZS(I,J,K)=EZS(I,J,K)+(HYS(I,J,K)-HYS(I-1,J,K))*DTEDX
>     $                         -(HXS(I,J,K)-HXS(I,J-1,K))*DTEDY
1774c1750
< C     Subroutine HXSFLD modified
---
> C
1778d1753
<      IFDTD_STUDY=NX1-1
1783c1758
< C          DO 10 I=2,NX1
---
>          DO 10 I=2,NX1
1792c1767
< C        NON-MAGNETIC MATERIAL -- ***** modified code !!
---
> C        NON-MAGNETIC MATERIAL
1794,1802c1769,1770
< C 105      HXS(I,J,K)=HXS(I,J,K)-(EZS(I,J+1,K)-EZS(I,J,K))*DTMDY
< C      $                       +(EYS(I,J,K+1)-EYS(I,J,K))*DTMDZ
<  105     FDTD_STUDY1=HXS(2,J,K)
<      FDTD_STUDY2=EZS(2,J+1,K)
<      FDTD_STUDY3=EZS(2,J,K)
<      FDTD_STUDY4=DTMDY
<      FDTD_STUDY5=EYS(2,J,K+1)
<      FDTD_STUDY6=EYS(2,J,K)
<      FDTD_STUDY7=DTMDZ
---
> 105      HXS(I,J,K)=HXS(I,J,K)-(EZS(I,J+1,K)-EZS(I,J,K))*DTMDY
>     $                         +(EYS(I,J,K+1)-EYS(I,J,K))*DTMDZ
1837c1805
< C     Subroutine HYSFLD modified
---
> C
1841d1808
<      IFDTD_STUDY=NX1
1846c1813
< C          DO 10 I=1,NX1
---
>          DO 10 I=1,NX1
1855c1822
< C        NON-MAGNETIC MATERIAL -- ***** modified code !!!!
---
```

```
> C       NON-MAGNETIC MATERIAL
1857,1865c1824,1825
< C 105     HYS(I,J,K)=HYS(I,J,K)-(EXS(I,J,K+1)-EXS(I,J,K))*DTMDZ
< C    $                   +(EZS(I+1,J,K)-EZS(I,J,K))*DTMDX
<  105    FDTD_STUDY1=HYS(1,J,K+1)
<     FDTD_STUDY2=EXS(1,J,K)
<     FDTD_STUDY3=EXS(1,J,K)
<     FDTD_STUDY4=DTMDZ
<     FDTD_STUDY5=EZS(2,J,K)
<     FDTD_STUDY6=EZS(1,J,K)
<     FDTD_STUDY7=DTMDX
---
>  105     HYS(I,J,K)=HYS(I,J,K)-(EXS(I,J,K+1)-EXS(I,J,K))*DTMDZ
>     $                   +(EZS(I+1,J,K)-EZS(I,J,K))*DTMDX
1900c1860
< C    Subroutine HZSFLD modified
---
> C
1904d1863
<      IFDTD_STUDY=NX1
1909c1868
< C        DO 10 I=1,NX1
---
>         DO 10 I=1,NX1
1918c1877
< C       NON-MAGNETIC MATERIAL -- ***** modified code !!!
---
> C       NON-MAGNETIC MATERIAL
1920,1928c1879,1880
< C 105     HZS(I,J,K)=HZS(I,J,K)-(EYS(I+1,J,K)-EYS(I,J,K))*DTMDX
< C    $                   +(EXS(I,J+1,K)-EXS(I,J,K))*DTMDY
<  105    FDTD_STUDY1=HZS(1,J,K)
<     FDTD_STUDY2=EYS(2,J,K)
<     FDTD_STUDY3=EYS(1,J,K)
<     FDTD_STUDY4=DTMDX
<     FDTD_STUDY5=EXS(1,J+1,K)
<     FDTD_STUDY6=EXS(1,J,K)
<     FDTD_STUDY7=DTMDY
---
>  105     HZS(I,J,K)=HZS(I,J,K)-(EYS(I+1,J,K)-EYS(I,J,K))*DTMDX
>     $                   +(EXS(I,J+1,K)-EXS(I,J,K))*DTMDY
1963c1915
< C    Subroutine RADHXZ modified
---
> C
1989,1991d1940
< C ------------------- ******************** modified here !!!!
< C
<      IFDTD_STUDY=NX1-3
1993,2017c1942,1954
< C     DO 102 I=3,NX1-1
< C      HXS(I,J,1)=-HXSZ2(I,J,2)+CZD*(HXS(I,J,2)+HXSZ2(I,J,1))
< C     2+CZZ*(HXSZ1(I,J,1)+HXSZ1(I,J,2))
< C     3+CZFXD*(HXSZ1(I+1,J,1)-2.*HXSZ1(I,J,1)+HXSZ1(I-1,J,1)
< C     4       +HXSZ1(I+1,J,2)-2.*HXSZ1(I,J,2)+HXSZ1(I-1,J,2))
< C     3+CZFYD*(HXSZ1(I,J+1,1)-2.*HXSZ1(I,J,1)+HXSZ1(I,J-1,1)
< C     4       +HXSZ1(I,J+1,2)-2.*HXSZ1(I,J,2)+HXSZ1(I,J-1,2))
< C      HXS(I,J,NZ1)=-HXSZ2(I,J,3)+CZU*(HXS(I,J,NZ-2)+HXSZ2(I,J,4))
< C     2+CZZ*(HXSZ1(I,J,4)+HXSZ1(I,J,3))
< C     3+CZFXD*(HXSZ1(I+1,J,4)-2.*HXSZ1(I,J,4)+HXSZ1(I-1,J,4)
< C     4       +HXSZ1(I+1,J,3)-2.*HXSZ1(I,J,3)+HXSZ1(I-1,J,3))
< C     3+CZFYD*(HXSZ1(I,J+1,4)-2.*HXSZ1(I,J,4)+HXSZ1(I,J-1,4)
< C     4       +HXSZ1(I,J+1,3)-2.*HXSZ1(I,J,3)+HXSZ1(I,J-1,3))
<      FDTD_STUDY1=HXS(3,J,2)
<      FDTD_STUDY2=HXSZ2(3,J,1)
<      FDTD_STUDY3=HXSZ1(2,J,1)
<      FDTD_STUDY4=HXSZ1(3,J+1,1)
<      FDTD_STUDY5=HXSZ1(3,J-1,1)
<      FDTD_STUDY6=HXSZ2(3,J,3)
<      FDTD_STUDY7=HXS(3,J,1)
<      FDTD_STUDY8=HXS(3,J,NZ-2)
<      FDTD_STUDY9=HXS(3,J,1)
<      FDTD_STUDY10=HXSZ1(2,J,3)
<      FDTD_STUDY11=HXSZ1(3,J+1,3)
<      FDTD_STUDY12=HXSZ1(3,J-1,3)
---
>      DO 102 I=3,NX1-1
>      HXS(I,J,1)=-HXSZ2(I,J,2)+CZD*(HXS(I,J,2)+HXSZ2(I,J,1))
>     2+CZZ*(HXSZ1(I,J,1)+HXSZ1(I,J,2))
>     3+CZFXD*(HXSZ1(I+1,J,1)-2.*HXSZ1(I,J,1)+HXSZ1(I-1,J,1)
>     4       +HXSZ1(I+1,J,2)-2.*HXSZ1(I,J,2)+HXSZ1(I-1,J,2))
>     3+CZFYD*(HXSZ1(I,J+1,1)-2.*HXSZ1(I,J,1)+HXSZ1(I,J-1,1)
>     4       +HXSZ1(I,J+1,2)-2.*HXSZ1(I,J,2)+HXSZ1(I,J-1,2))
>      HXS(I,J,NZ1)=-HXSZ2(I,J,3)+CZU*(HXS(I,J,NZ-2)+HXSZ2(I,J,4))
>     2+CZZ*(HXSZ1(I,J,4)+HXSZ1(I,J,3))
>     3+CZFXD*(HXSZ1(I+1,J,4)-2.*HXSZ1(I,J,4)+HXSZ1(I-1,J,4)
>     4       +HXSZ1(I+1,J,3)-2.*HXSZ1(I,J,3)+HXSZ1(I-1,J,3))
>     3+CZFYD*(HXSZ1(I,J+1,4)-2.*HXSZ1(I,J,4)+HXSZ1(I,J-1,4)
>     4       +HXSZ1(I,J+1,3)-2.*HXSZ1(I,J,3)+HXSZ1(I,J-1,3))
2038c1975
< C    Subroutine RADHYX modified
---
> C
2064,2067d2000
< C ----------------------************** modified here !!!!!!
< C
<      IFDTD_STUDY=NY1-3
<      JFDTD_STUDY=NY1
2069,2093c2002,2014
< C      DO 102 J=3,NY1-1
< C      HYS(1,J,K)=-HYSX2(2,J,K)+CXD*(HYS(2,J,K)+HYSX2(1,J,K))
< C     2+CXX*(HYSX1(1,J,K)+HYSX1(2,J,K))
< C     3+CXFYD*(HYSX1(1,J+1,K)-2.*HYSX1(1,J,K)+HYSX1(1,J-1,K)
< C     4       +HYSX1(2,J+1,K)-2.*HYSX1(2,J,K)+HYSX1(2,J-1,K))
< C     3+CXFZD*(HYSX1(1,J,K+1)-2.*HYSX1(1,J,K)+HYSX1(1,J,K-1)
< C     4       +HYSX1(2,J,K+1)-2.*HYSX1(2,J,K)+HYSX1(2,J,K-1))
< C      HYS(NX1,J,K)=-HYSX2(3,J,K)+CXU*(HYS(NX-2,J,K)+HYSX2( 4,J,K))
< C     2+CXX*(HYSX1(4,J,K)+HYSX1(3,J,K))
< C     3+CXFYD*(HYSX1(4,J+1,K)-2.*HYSX1(4,J,K)+HYSX1(4,J-1,K)
< C     4       +HYSX1(3,J+1,K)-2.*HYSX1(3,J,K)+HYSX1(3,J-1,K))
< C     3+CXFZD*(HYSX1(4,J,K+1)-2.*HYSX1(4,J,K)+HYSX1(4,J,K-1)
< C     4       +HYSX1(3,J,K+1)-2.*HYSX1(3,J,K)+HYSX1(3,J,K-1))
<      FDTD_STUDY1=HYS(2,3,K)
<      FDTD_STUDY2=HYSX2(1,3,K)
<      FDTD_STUDY3=HYSX1(1,2,K)
<      FDTD_STUDY4=HYSX1(1,3,K+1)
<      FDTD_STUDY5=HYSX1(1,3,K-1)
<      FDTD_STUDY6=HYS(1,3,K)
<      FDTD_STUDY7=HYSX2(3,3,K)
<      FDTD_STUDY8=HYS(NX-2,3,K)
<      FDTD_STUDY9=HYS(NX1,3,K)
<      FDTD_STUDY10=HYSX1(3,2,K)
<      FDTD_STUDY11=HYSX1(3,3,K+1)
<      FDTD_STUDY12=HYSX1(3,3,K-1)
---
>      DO 102 J=3,NY1-1
>      HYS(1,J,K)=-HYSX2(2,J,K)+CXD*(HYS(2,J,K)+HYSX2(1,J,K))
>     2+CXX*(HYSX1(1,J,K)+HYSX1(2,J,K))
>     3+CXFYD*(HYSX1(1,J+1,K)-2.*HYSX1(1,J,K)+HYSX1(1,J-1,K)
>     4       +HYSX1(2,J+1,K)-2.*HYSX1(2,J,K)+HYSX1(2,J-1,K))
>     3+CXFZD*(HYSX1(1,J,K+1)-2.*HYSX1(1,J,K)+HYSX1(1,J,K-1)
>     4       +HYSX1(2,J,K+1)-2.*HYSX1(2,J,K)+HYSX1(2,J,K-1))
>      HYS(NX1,J,K)=-HYSX2(3,J,K)+CXU*(HYS(NX-2,J,K)+HYSX2( 4,J,K))
>     2+CXX*(HYSX1(4,J,K)+HYSX1(3,J,K))
>     3+CXFYD*(HYSX1(4,J+1,K)-2.*HYSX1(4,J,K)+HYSX1(4,J-1,K)
>     4       +HYSX1(3,J+1,K)-2.*HYSX1(3,J,K)+HYSX1(3,J-1,K))
>     3+CXFZD*(HYSX1(4,J,K+1)-2.*HYSX1(4,J,K)+HYSX1(4,J,K-1)
>     4       +HYSX1(3,J,K+1)-2.*HYSX1(3,J,K)+HYSX1(3,J,K-1))
2110d2030
<      JFDTD_STUDY=1
```

```
2115c2035
< C     Subroutine RADHZY modified
---
> C
2141d2060
<       IFDTD_STUDY=NX-5
2143,2167c2062,2074
< C       DO 102 I=3,NX-3
< C       HZS(I,1,K)=-HZSY2(I,2,K)+CYD*(HZS(I,2,K)+HZSY2(I,1,K))
< C      2+CYY*(HZSY1(I,1,K)+HZSY1(I,2,K))
< C      3+CYFXD*(HZSY1(I+1,1,K)-2.*HZSY1(I,1,K)+HZSY1(I-1,1,K)
< C      4       +HZSY1(I+1,2,K)-2.*HZSY1(I,2,K)+HZSY1(I-1,2,K))
< C      3+CYFZD*(HZSY1(I,1,K+1)-2.*HZSY1(I,1,K)+HZSY1(I,1,K-1)
< C      4       +HZSY1(I,2,K+1)-2.*HZSY1(I,2,K)+HZSY1(I,2,K-1))
< C       HZS(I,NY1,K)=-HZSY2(I,3,K)+CYU*(HZS(I,NY-2,K)+HZSY2(I,4,K))
< C      2+CYY*(HZSY1(I,4,K)+HZSY1(I,3,K))
< C      3+CYFXD*(HZSY1(I+1,4,K)-2.*HZSY1(I,4,K)+HZSY1(I-1,4,K)
< C      4       +HZSY1(I+1,3,K)-2.*HZSY1(I,3,K)+HZSY1(I-1,3,K))
< C      3+CYFZD*(HZSY1(I,4,K+1)-2.*HZSY1(I,4,K)+HZSY1(I,4,K-1)
< C      4       +HZSY1(I,3,K+1)-2.*HZSY1(I,3,K)+HZSY1(I,3,K-1))
<       FDTD_STUDY1=HZSY2(3,1,K)
<       FDTD_STUDY2=HZSY1(2,1,K)
<       FDTD_STUDY3=HZSY1(3,1,K+1)
<       FDTD_STUDY4=HZSY1(3,1,K-1)
<       FDTD_STUDY5=HZS(3,1,K)
<       FDTD_STUDY6=HZSY2(3,3,K)
<       FDTD_STUDY7=HZS(3,NY-2,K)
<       FDTD_STUDY8=HZSY1(2,3,K)
<       FDTD_STUDY9=HZSY1(4,3,K)
<       FDTD_STUDY10=HZSY1(3,3,K+1)
<       FDTD_STUDY11=HZSY1(3,3,K-1)
<       FDTD_STUDY12=HZS(3,NY1,K)
---
>       DO 102 I=3,NX-3
>       HZS(I,1,K)=-HZSY2(I,2,K)+CYD*(HZS(I,2,K)+HZSY2(I,1,K))
>      2+CYY*(HZSY1(I,1,K)+HZSY1(I,2,K))
>      3+CYFXD*(HZSY1(I+1,1,K)-2.*HZSY1(I,1,K)+HZSY1(I-1,1,K)
>      4       +HZSY1(I+1,2,K)-2.*HZSY1(I,2,K)+HZSY1(I-1,2,K))
>      3+CYFZD*(HZSY1(I,1,K+1)-2.*HZSY1(I,1,K)+HZSY1(I,1,K-1)
>      4       +HZSY1(I,2,K+1)-2.*HZSY1(I,2,K)+HZSY1(I,2,K-1))
>       HZS(I,NY1,K)=-HZSY2(I,3,K)+CYU*(HZS(I,NY-2,K)+HZSY2(I,4,K))
>      2+CYY*(HZSY1(I,4,K)+HZSY1(I,3,K))
>      3+CYFXD*(HZSY1(I+1,4,K)-2.*HZSY1(I,4,K)+HZSY1(I-1,4,K)
>      4       +HZSY1(I+1,3,K)-2.*HZSY1(I,3,K)+HZSY1(I-1,3,K))
>      3+CYFZD*(HZSY1(I,4,K+1)-2.*HZSY1(I,4,K)+HZSY1(I,4,K-1)
>      4       +HZSY1(I,3,K+1)-2.*HZSY1(I,3,K)+HZSY1(I,3,K-1))
2188c2095
< C     Subroutine RADHXY modified
---
> C
2214,2216d2120
< C ------------------************** modified here!!!!!!!!!!!
< C
<       IFDTD_STUDY=NX1-3
2218,2242c2122,2134
< C       DO 102 I=3,NX1-1
< C       HXS(I,1,K)=-HXSY2(I,2,K)+CYD*(HXS(I,2,K)+HXSY2(I,1,K))
< C      2+CYY*(HXSY1(I,1,K)+HXSY1(I,2,K))
< C      3+CYFXD*(HXSY1(I+1,1,K)-2.*HXSY1(I,1,K)+HXSY1(I-1,1,K)
< C      4       +HXSY1(I+1,2,K)-2.*HXSY1(I,2,K)+HXSY1(I-1,2,K))
< C      3+CYFZD*(HXSY1(I,1,K+1)-2.*HXSY1(I,1,K)+HXSY1(I,1,K-1)
< C      4       +HXSY1(I,2,K+1)-2.*HXSY1(I,2,K)+HXSY1(I,2,K-1))
< C       HXS(I,NY1,K)=-HXSY2(I,3,K)+CYU*(HXS(I,NY-2,K)+HXSY2(I,4,K))
< C      2+CYY*(HXSY1(I,4,K)+HXSY1(I,3,K))
< C      3+CYFXD*(HXSY1(I+1,4,K)-2.*HXSY1(I,4,K)+HXSY1(I-1,4,K)
< C      4       +HXSY1(I+1,3,K)-2.*HXSY1(I,3,K)+HXSY1(I-1,3,K))
< C      3+CYFZD*(HXSY1(I,4,K+1)-2.*HXSY1(I,4,K)+HXSY1(I,4,K-1)
< C      4       +HXSY1(I,3,K+1)-2.*HXSY1(I,3,K)+HXSY1(I,3,K-1))
<       FDTD_STUDY1=HXSY2(3,1,K)
<       FDTD_STUDY2=HXSY1(2,1,K)
<       FDTD_STUDY3=HXSY1(3,1,K+1)
<       FDTD_STUDY4=HXSY1(3,1,K-1)
<       FDTD_STUDY5=HXS(3,1,K)
<       FDTD_STUDY6=HXSY2(3,3,K)
<       FDTD_STUDY7=HXS(3,NY-2,K)
<       FDTD_STUDY8=HXSY1(4,3,K)
<       FDTD_STUDY9=HXSY1(2,3,K)
<       FDTD_STUDY10=HXSY1(3,3,K+1)
<       FDTD_STUDY11=HXSY1(3,3,K-1)
<       FDTD_STUDY12=HXS(3,NY1,K)
---
>       DO 102 I=3,NX1-1
>       HXS(I,1,K)=-HXSY2(I,2,K)+CYD*(HXS(I,2,K)+HXSY2(I,1,K))
>      2+CYY*(HXSY1(I,1,K)+HXSY1(I,2,K))
>      3+CYFXD*(HXSY1(I+1,1,K)-2.*HXSY1(I,1,K)+HXSY1(I-1,1,K)
>      4       +HXSY1(I+1,2,K)-2.*HXSY1(I,2,K)+HXSY1(I-1,2,K))
>      3+CYFZD*(HXSY1(I,1,K+1)-2.*HXSY1(I,1,K)+HXSY1(I,1,K-1)
>      4       +HXSY1(I,2,K+1)-2.*HXSY1(I,2,K)+HXSY1(I,2,K-1))
>       HXS(I,NY1,K)=-HXSY2(I,3,K)+CYU*(HXS(I,NY-2,K)+HXSY2(I,4,K))
>      2+CYY*(HXSY1(I,4,K)+HXSY1(I,3,K))
>      3+CYFXD*(HXSY1(I+1,4,K)-2.*HXSY1(I,4,K)+HXSY1(I-1,4,K)
>      4       +HXSY1(I+1,3,K)-2.*HXSY1(I,3,K)+HXSY1(I-1,3,K))
>      3+CYFZD*(HXSY1(I,4,K+1)-2.*HXSY1(I,4,K)+HXSY1(I,4,K-1)
>      4       +HXSY1(I,3,K+1)-2.*HXSY1(I,3,K)+HXSY1(I,3,K-1))
2263c2155
< C     Subroutine RADHYZ modified
---
> C
2289,2291d2180
< C--------------**************** modified here !!!!!!!
< C
<       IFDTD_STUDY=NX-3
2293,2317c2182,2194
< C       DO 102 I=2,NX-2
< C       HYS(I,J,1)=-HYSZ2(I,J,2)+CZD*(HYS(I,J,2)+HYSZ2(I,J,1))
< C      2+CZZ*(HYSZ1(I,J,1)+HYSZ1(I,J,2))
< C      3+CZFXD*(HYSZ1(I+1,J,1)-2.*HYSZ1(I,J,1)+HYSZ1(I-1,J,1)
< C      4       +HYSZ1(I+1,J,2)-2.*HYSZ1(I,J,2)+HYSZ1(I-1,J,2))
< C      3+CZFYD*(HYSZ1(I,J+1,1)-2.*HYSZ1(I,J,1)+HYSZ1(I,J-1,1)
< C      4       +HYSZ1(I,J+1,2)-2.*HYSZ1(I,J,2)+HYSZ1(I,J-1,2))
< C       HYS(I,J,NZ1)=-HYSZ2(I,J,3)+CZU*(HYS(I,J,NZ-2)+HYSZ2(I,J,4))
< C      2+CZZ*(HYSZ1(I,J,4)+HYSZ1(I,J,3))
< C      3+CZFXD*(HYSZ1(I+1,J,4)-2.*HYSZ1(I,J,4)+HYSZ1(I-1,J,4)
< C      4       +HYSZ1(I+1,J,3)-2.*HYSZ1(I,J,3)+HYSZ1(I-1,J,3))
< C      3+CZFYD*(HYSZ1(I,J+1,4)-2.*HYSZ1(I,J,4)+HYSZ1(I,J-1,4)
< C      4       +HYSZ1(I,J+1,3)-2.*HYSZ1(I,J,3)+HYSZ1(I,J-1,3))
<       FDTD_STUDY1=HYSZ2(2,J,1)
<       FDTD_STUDY2=HYSZ1(1,J,1)
<       FDTD_STUDY3=HYSZ1(2,J+1,1)
<       FDTD_STUDY4=HYSZ1(2,J-1,1)
<       FDTD_STUDY5=HYS(2,J,1)
<       FDTD_STUDY6=HYSZ2(2,J,3)
<       FDTD_STUDY7=HYS(2,J,NZ-2)
<       FDTD_STUDY8=HYSZ1(2,J,3)
<       FDTD_STUDY9=HYSZ1(1,J,3)
<       FDTD_STUDY10=HYSZ1(2,J+1,3)
<       FDTD_STUDY11=HYSZ1(2,J-1,3)
<       FDTD_STUDY12=HYS(2,J,NZ1)
---
>       DO 102 I=2,NX-2
>       HYS(I,J,1)=-HYSZ2(I,J,2)+CZD*(HYS(I,J,2)+HYSZ2(I,J,1))
>      2+CZZ*(HYSZ1(I,J,1)+HYSZ1(I,J,2))
>      3+CZFXD*(HYSZ1(I+1,J,1)-2.*HYSZ1(I,J,1)+HYSZ1(I-1,J,1)
>      4       +HYSZ1(I+1,J,2)-2.*HYSZ1(I,J,2)+HYSZ1(I-1,J,2))
>      3+CZFYD*(HYSZ1(I,J+1,1)-2.*HYSZ1(I,J,1)+HYSZ1(I,J-1,1)
>      4       +HYSZ1(I,J+1,2)-2.*HYSZ1(I,J,2)+HYSZ1(I,J-1,2))
>       HYS(I,J,NZ1)=-HYSZ2(I,J,3)+CZU*(HYS(I,J,NZ-2)+HYSZ2(I,J,4))
>      2+CZZ*(HYSZ1(I,J,4)+HYSZ1(I,J,3))
>      3+CZFXD*(HYSZ1(I+1,J,4)-2.*HYSZ1(I,J,4)+HYSZ1(I-1,J,4)
>      4       +HYSZ1(I+1,J,3)-2.*HYSZ1(I,J,3)+HYSZ1(I-1,J,3))
>      3+CZFYD*(HYSZ1(I,J+1,4)-2.*HYSZ1(I,J,4)+HYSZ1(I,J-1,4)
>      4      +HYSZ1(I,J+1,3)-2.*HYSZ1(I,J,3)+HYSZ1(I,J-1,3))
2338c2215
< C     Subroutine RADHZX modified
---
> C
2364,2367d2240
< C------------------------************** modified here !!!
< C
<        IFDTD_STUDY=NY-3
<        JFDTD_STUDY=NY
2369,2393c2242,2254
< C      DO 102 J=2,NY-2
< C      HZS(1,J,K)=-HZSX2(2,J,K)+CXD*(HZS(2,J,K)+HZSX2(1,J,K))
< C     2+CXX*(HZSX1(1,J,K)+HZSX1(2,J,K))
< C     3+CXFYD*(HZSX1(1,J+1,K)-2.*HZSX1(1,J,K)+HZSX1(1,J-1,K)
< C     4      +HZSX1(2,J+1,K)-2.*HZSX1(2,J,K)+HZSX1(2,J-1,K))
< C     3+CXFZD*(HZSX1(1,J,K+1)-2.*HZSX1(1,J,K)+HZSX1(1,J,K-1)
< C     4      +HZSX1(2,J,K+1)-2.*HZSX1(2,J,K)+HZSX1(2,J,K-1))
< C      HZS(NX1,J,K)=-HZSX2(3,J,K)+CXU*(HZS(NX-2,J,K)+HZSX2(4,J,K))
< C     2+CXX*(HZSX1(4,J,K)+HZSX1(3,J,K))
< C     3+CXFYD*(HZSX1(4,J+1,K)-2.*HZSX1(4,J,K)+HZSX1(4,J-1,K)
< C     4      +HZSX1(3,J+1,K)-2.*HZSX1(3,J,K)+HZSX1(3,J-1,K))
< C     3+CXFZD*(HZSX1(4,J,K+1)-2.*HZSX1(4,J,K)+HZSX1(4,J,K-1)
< C     4      +HZSX1(3,J,K+1)-2.*HZSX1(3,J,K)+HZSX1(3,J,K-1))
<        FDTD_STUDY1=HZSX2(1,2,K)
<        FDTD_STUDY2=HZSX1(1,1,K)
<        FDTD_STUDY3=HZSX1(1,2,K+1)
<        FDTD_STUDY4=HZSX1(1,2,K-1)
<        FDTD_STUDY5=HZS(1,2,1)
<        FDTD_STUDY6=HZSX2(3,2,K)
<        FDTD_STUDY7=HYS(NX-2,2,K)
<        FDTD_STUDY8=HZSX1(3,1,K)
<        FDTD_STUDY9=HZSX1(3,3,K)
<        FDTD_STUDY10=HZSX1(3,2,K+1)
<        FDTD_STUDY11=HZSX1(3,2,K-1)
<        FDTD_STUDY12=HZS(NX1,2,K)
---
>        DO 102 J=2,NY-2
>        HZS(1,J,K)=-HZSX2(2,J,K)+CXD*(HZS(2,J,K)+HZSX2(1,J,K))
>       2+CXX*(HZSX1(1,J,K)+HZSX1(2,J,K))
>       3+CXFYD*(HZSX1(1,J+1,K)-2.*HZSX1(1,J,K)+HZSX1(1,J-1,K)
>       4      +HZSX1(2,J+1,K)-2.*HZSX1(2,J,K)+HZSX1(2,J-1,K))
>       3+CXFZD*(HZSX1(1,J,K+1)-2.*HZSX1(1,J,K)+HZSX1(1,J,K-1)
>       4      +HZSX1(2,J,K+1)-2.*HZSX1(2,J,K)+HZSX1(2,J,K-1))
>        HZS(NX1,J,K)=-HZSX2(3,J,K)+CXU*(HZS(NX-2,J,K)+HZSX2(4,J,K))
>       2+CXX*(HZSX1(4,J,K)+HZSX1(3,J,K))
>       3+CXFYD*(HZSX1(4,J+1,K)-2.*HZSX1(4,J,K)+HZSX1(4,J-1,K)
>       4      +HZSX1(3,J+1,K)-2.*HZSX1(3,J,K)+HZSX1(3,J-1,K))
>       3+CXFZD*(HZSX1(4,J,K+1)-2.*HZSX1(4,J,K)+HZSX1(4,J,K-1)
>       4      +HZSX1(3,J,K+1)-2.*HZSX1(3,J,K)+HZSX1(3,J,K-1))
2410d2270
<        JFDTD_STUDY=1
```

Appendix F -- Radiation Boundary Condition Microcode

```
;================================================================================
;   TITLE:          boundary
;
;   VERSION:        1.0
;
;   DATE:           11 Nov 91
;
;   AUTHOR:         Raley Marek
;
;   PURPOSE:        Computes the value of the 2nd order Mur boundary
;                   equation for the method of finite-difference
;                   time domian. Using double precision math, this
;                   code achieves roughly 50 MFLOPS when operating
;                   on long vectors of data.
;
;   REGISTERS:      R1, R2, R4, R5, R7, R8, R9, R11, R12, R13,
;                   ACCA, ACCB, MBR. MAR, STAT
;
;   POINTERS:       APT, BPT, CPT, DPT, AIN, BIN, CIN, DIN, IN3
;
;   LINES:          33
;
;   LANGUAGE:       FPASP Microcode Assembler Version 4.7
;
;   HISTORY:        11 Nov 91 - Code written for thesis - jrm
;
;================================================================================

;---------------------------------------------------------------------------
;
;   1   Clear the MAR and R1 upper.
;
;       XOR precharged buses (w/ no drivers so both buses all ones).
;       This loads zeros into shifter. Shift put 1 in upper bit, so
;       R1 lower has all zeros except for highest order bit.
;
;---------------------------------------------------------------------------

R1=CU  R1=CL  XORL   GNDCU  SR1L   MAR=CU;


;---------------------------------------------------------------------------
;
;   2   MBR upper = START ADDR
;
;       MBR lower = ITERATIONS
;
;       AIN, CIN = 2
;
;       Floating point unit set to double precision by flipped R1.
;
;       Increment & left shift R1 upper which loads "2" into AIN & CIN.
;
;---------------------------------------------------------------------------

AU=R1  BU=R1  BL=R1  C_TIE  FLIPB  MBR=D  FP+LDF INCU   SL0U   AIN=CU
CIN=CL MAR+2  READ   BACT;
```

```
;------------------------------------------------------------------------
;
;     3   MAR, BPT = STA'.T
;
;         IN3 = -(ITERATIONS)
;
;------------------------------------------------------------------------

AU=MBR AL=MBR BPT=CU IN3=CL MOVNU  NEGAL  PASSU  PASSL  MAR=CU;

;------------------------------------------------------------------------
;
;     4   MBR = K1
;
;         BIN, CIN = 10 (ten)
;
;------------------------------------------------------------------------

C_TIE  MBR=D  MOVNU  PASSU  BIN=CU DIN=CL MAR+2  READ   BACT   ILZU
#0000000000001010;

;------------------------------------------------------------------------
;
;     5   MBR = K2
;
;         R7 = K1
;
;         BPT = START + 10
;
;------------------------------------------------------------------------

CD     C=MBR  R7=CU  R7=CL  MBR=D  BPT+B  MAR+2  READ   BACT;

;------------------------------------------------------------------------
;
;     6   MBR = K3
;
;         R1 = 0
;
;         R8 = K2
;
;         BPT = START + 20
;
;------------------------------------------------------------------------

CD     C=MBR  R8=CU  R8=CL  MBR=D  BPT+B  MAR+2  READ   BACT;

;------------------------------------------------------------------------
;
;     7   MBR lower = UP
;
;         MBR upper = DOWN
;
;         R9 = K3
;
;         BPT = START + 22
;
;------------------------------------------------------------------------

BU=MBR BL=MBR CD     C=MBR  R9=CU  R9=CL  MBR=D  BPT+A
MAR+2  READ   BACT;
```

```
;------------------------------------------------------------------------
;
;   8    MBR lower = OUT
;
;        APT = UP
;
;        CPT = DOWN
;
;------------------------------------------------------------------------

CD    C=MBR  APT=CU CPT=CL MBR=D  MAR+2  READ   BACT;

;------------------------------------------------------------------------
;
;   9    MBR = En(0,j-1)
;
;        DPT = DOWN
;
;        BPT = START + 24
;
;------------------------------------------------------------------------

CD    C=MBR  DPT=CL MBR=D  BPT+A  MAR+2  READ   BACT;

;------------------------------------------------------------------------
;
;   10   MBR = En(1,j-1)
;
;        ACCB = En(0,j-1)
;
;------------------------------------------------------------------------

BU=MBR BL=MBR MBR=D  ACCB=BBUS    MAR+2  READ   BACT;

;------------------------------------------------------------------------
;
;   11   R1 = En(0,j)
;
;------------------------------------------------------------------------

BU=MBR BL=MBR R1=D   FP++   a=ACCB b=BBUS MAR+2  READ   BACT;

;------------------------------------------------------------------------
;
;   12   R2 = En(1,j)
;
;------------------------------------------------------------------------

R2=D   MAR+2  READ   B^ACT;

;------------------------------------------------------------------------
;
;   13   MBR = En+1(1,j)
;
;        ACCA = En(0,j-1) + En(1,j-1)
;
;------------------------------------------------------------------------

BU=R2  BL=R2  CD    C=R1   MBR=D  FP++   a=CBUS b=BBUS ACCA=FP+
MAR+2  READ   BACT;
```

```
;--------------------------------------------------------------------------
;
;   14  R1 = En-1(0,j)
;
;       ACCB = En+1(1,j)
;
;--------------------------------------------------------------------------

BU=MBR BL=MBR  R1=D   ACCB=BBUS    MAR+2  READ   BACT;


;--------------------------------------------------------------------------
;
;   15  R2 = En-1(1,j)
;
;       ACCB = En(0,j) + En(1,j)
;
;       MAR = UP
;
;--------------------------------------------------------------------------

AU=R8  AL=R8  BU=FP+ BL=FP+ CD     C=R1   R2=D   FP*    FP++   a=CBUS b=ACCB
ACFP+B ACCB=FP+      MAR=E  E=APT  READ   BACT;


;--------------------------------------------------------------------------
;
;   16  R1 = En(0,j,k+1.5)
;
;       ACCB = En-1(1,j)
;
;       R4 = En(0,j)+En(1,j)
;
;--------------------------------------------------------------------------

BU=R2  BL=R2  CD     C=FP+  R4=CU  R4=CL  R1=D   BCFP+B ACCB=BBUS    MAR+2
READ   BACT;


;--------------------------------------------------------------------------
;
;   17  R2  = En(1,j,k+1.5)
;
;       ACCB = En+1(1,j) + En-1(0,j)
;
;       APT = UP + 10
;
;       CPT = DOWN + 10
;
;       MAR = DOWN
;
;--------------------------------------------------------------------------

CD     C=FP*  R2=D   FP--   a=CBUS b=ACCB ACCB=FP+      APT+B  CPT+D  MAR=E
E=CPT  READ   BACT;


;--------------------------------------------------------------------------
;
;   18  R1 = En(1,j,k-.5)
;
;--------------------------------------------------------------------------

BU=R1  BL=R1  CD     C=R2   R1=D   FP++   a=CBUS b=BBUS MAR+2  READ   BACT;
```

```
;--------------------------------------------------------------------------
;
;   19  R2 = En(0,j,k-.5)
;
;       MAR = START+24
;
;       R5 = K2*[En(0,j)+En(1,j)] - En-1(1,j)
;
;--------------------------------------------------------------------------

AU=R7  AL=R7  BU=FP+ BL=FP+ CD    C=FP+  R5=CU  R5=CL  R2=D   FP*    BBFP+C
MAR=E  E=BPT  READ   BACT;


;--------------------------------------------------------------------------
;
;   20  R1 = En(0,j+1)
;
;       BPT = START + 26
;
;       ACCB = En(0,j,k+1.5) + En(1,j,k+1.5)
;
;--------------------------------------------------------------------------

BU=R1  BL=R1  CD    C=R2   R1=D   FP++   a=CBUS b=BBUS ACCB=FP+     BPT+A
MAR+2  READ   BACT;


;--------------------------------------------------------------------------
;
;   21  R2 = En(0,j+1)
;
;       ACCA = K1 * [ En-1(1,j) + En+1(0,j) ]
;
;       BPT = START + 28
;
;--------------------------------------------------------------------------

R2=D   FP++   a=ACCA b=ACCB ACCA=FP*     BPT+A  MAR+2  READ   BACT;


;--------------------------------------------------------------------------
;
;   22  MAR = START + 28
;
;       ACCB = En(0,j,k-.5)+En(1,j,k-.5)
;
;--------------------------------------------------------------------------

BU=R2  BL=R2  CD    C=R1   FP++   a=CBUS b=BBUS ACCB=FP+     MAR=E  E=BPT;
```

```
;-----------------------------------------------------------------------
;
;           Parameters inside the loop may be a function of N,
;
;           where N is the number of times through the loop,
;
;           starting at 1.  N must be less than INCREMENT.
;
;-----------------------------------------------------------------------

LOOP:

;-----------------------------------------------------------------------
;
;     23  MBR = En+1(0,j+N)
;
;         R13 = En(0,j+N-1) + En(1,j+N-1)
;
;         BPT = START + 30
;
;         ACCB= K2*[En(0,j+N-1)+En(1,j+N-1)] - En-1(1,j+N-1)
;
;-----------------------------------------------------------------------

BU=R5  BL=R5  CD    C=R4  R13=CU R13=CL MBR=D  FP++   a=FP+  b=ACCB
ACCB=BBUS    BPT+A  MAR+2  READ   BACT;


;-----------------------------------------------------------------------
;
;     24  R1 = En-1(0,j+N)
;
;         ACCB = En(0,j+N)+En(1,j+N)
;
;         R4 = En(0,j+N)+En(1,j+N)
;
;         BPT = START + 32
;
;-----------------------------------------------------------------------

CD    C=FP+  R4=CU  R4=CL  R1=D   FP++   a=ACCA b=ACCB ACCB=FP+      BPT+A
MAR+2  READ   BACT;


;-----------------------------------------------------------------------
;
;     25  R2 = En-1(1,j+N)
;
;         ACCA = En+1(0,j+N)
;
;         MAR = UP + 10
;
;-----------------------------------------------------------------------

AU=R8  AL=R8  BU=FP+ BL=FP+ CD     C=MBR  R2=D   FP*    FP++   a=FP+  b=ACCB
BBFP+C ACCA=CBUS     MAR=E  E=APT  READ   BACT;
```

```
;------------------------------------------------------------------------
;
;   26  R1 = En(0,j+N,k+1.5)
;
;       ACCA = En-1(1,j+N)
;
;       ACCB =     K1*[ En+1(1,j+N-1) + En-1(0,j+N-1) ]
;               +  K2*[ En(0,j+N-1) + En((1,j+N-1) ]
;               -  En-1(1,j+N-1)
;
;       BPT = START + 34
;
;------------------------------------------------------------------------

BU=R1  BL=R1  CD    C=R2  R1=D  FP++  a=ACCA b=BBUS ACCA=CBUS
ACCB=FP+     BPT+A  MAR+2  READ   BACT;


;------------------------------------------------------------------------
;
;   27  R2 = En(1,j+N,k+1.5)
;
;       MAR = DOWN + 10*N
;
;       APT = UP + 10*(N+1)
;
;       CPT = DOWN + 10*(N+1)
;
;------------------------------------------------------------------------

AU=R9  AL=R9  BU=FP+ BL=FP+ CD    C=FP*  R2=D  FP*   FP--  a=CBUS
b=ACCA ACFP+B APT+B  CPT+D  MAR=E  E=CPT  READ   BACT;


;------------------------------------------------------------------------
;
;   28  R1 = En(0,j+N,k-.5)
;
;       ACCA = En-1(0,j+N)+En+1(0,j+N)
;
;------------------------------------------------------------------------

BU=R2  BL=R2  CD    C=R1   R1=D   FP++   a=CBUS b=BBUS ACCA=FP+      MAR+2
READ   BACT;


;------------------------------------------------------------------------
;
;   29  R2 = En(1,j+N,k-.5)
;
;       IN3 = N - INCREMENT
;
;       MAR= START+24+10*N
;
;       R5 = K2*[ En(0,j+N)+En(1,j+N) ] - En-1(1,j)
;
;------------------------------------------------------------------------

AU=R7  AL=R7  BU=FP+ BL=FP+ CD    C=FP+  R5=CU  R5=CL  R2=D   IN3+   FP*
FP++   a=ACCB b=FP*  MAR=E  E=BPT  READ   BACT;
```

```
;---------------------------------------------------------------------------
;
;   30  R1 = En(0,j+1+N)
;
;       ACCB = En(0,j+N,k+1.5)+En(1,j+N,k+1.5)
;
;       BPT = START+26 + 10*N
;
;---------------------------------------------------------------------------

BU=R1 BL=R1 CD    C=R2  R1=D  FP++  a=CBUS b=BBUS ACCB=FP+      BPT+A
MAR+2 READ  BACT;


;---------------------------------------------------------------------------
;
;   31  R2 = En(1,j+1+N)
;
;       MBR = En+1(0,j+N-1) -- (Final Answer N)
;
;       ACCA= K1 * (En-1 + En+1)
;
;       MAR= OUT+10*(N-1)
;
;       BPT = START+28 + 10*N
;
;---------------------------------------------------------------------------

BU=R13 BL=R13 CD    C=FP+  MBR=CU MBR=CL R2=D  FP++  a=ACCB b=BBUS ACCA=FP*
BPT+A  MAR=E  E=DPT READ  BACT  BR    IN3NZ LOOP;


;---------------------------------------------------------------------------
;
;   32  MAR= START+28+10*N
;
;       ACCB = En(0,j+N,k-.5)+En(1,j+N,k-.5)
;
;       DPT= OUT + 10*N
;
;---------------------------------------------------------------------------

BU=R2  BL=R2  CD    C=R1  FP++  a=CBUS b=BBUS ACCB=FP+
DPT+D  MAR=E  E=BPT WRITE BACT;


END:


;---------------------------------------------------------------------------
;
;   33  Set done status bit
;
;---------------------------------------------------------------------------

AU=R1  BU=R1  STAT=CU     XORU  SL1U;
```

Appendix G -- Initial Data for FPASP

This is an annotated listing of U0_LOADMOD and L0_LOADMOD. Since the output will be in hex, the expected result is entered into these data tables as well, to make comparison easier. Upper memory and lower memory are both 32 bits wide. Addresses are in hex. "H" stands for hex, and "D" stands for double-precision. Note that the double precision numbers span both upper and lower memory blocks. Comments in "{}" and indented show other parts of data structure not used by this simulation.

Address	Upper Memory	Lower Memory	COMMENTS
0	H 0000002E	H 00000003	Pointer to START, # of Iterations
2	H 00000000	H 00000000	
4	H 00000000	H 00000000	
6	H 00000000	H 00000000	
8	H 00000000	H 00000000	{K1} [Previous data vector]
A	H 00000000	H 00000000	{K2}
C	H 00000000	H 00000000	{K3}
E	H 00000000	H 00000000	{UP,DOWN}
10	H 00000000	H 00000000	{(Blank),OUT}
12	H 00000000	H 00000000	{E^{n}(0,j-1,k-½)}
14	H 00000000	H 00000000	{E^{n}(1.j-1,k-½)}
16	D 108.6	D 108.6	E^{n}(0,j,k-½) [Address UP]
18	D 7.6	D 7.6	E^{n}(1,j,k-½)
1A	H 00000000	H 00000000	{E^{n+1}(0,j,k-½)}
1C	H 00000000	H 00000000	{E^{n-1}(0,j,k-½)}
1E	H 00000000	H 00000000	{E^{n-1}(1,j,k-½)}
20	D 4.3	D 4.3	E^{n}(0,j+1,k-½)
22	D 3.3	D 3.3	E^{n}(1,j+1,k-½)
24	H 00000000	H 00000000	{E^{n+1}(0,j+1,k-½)}
26	H 00000000	H 00000000	{E^{n-1}(0,j+1,k-½)}
28	H 00000000	H 00000000	{E^{n-1}(1,j+1,k-½)}
2A	D 1.3	D 1.3	E^{n}(0,j+2,k-½)
2C	D 2.3	D 2.3	E^{n}(1,j+2,k-½)
2E	D 1.2	D 1.2	K1 [START of present data vector]
30	D 1.4	D 1.4	K2
32	D 0.6	D 0.6	K3
34	H 00000016	H 00000072	UP, DOWN
36	H 00000000	H 0000008A	(Blank), OUT
38	D 5.3	D 5.3	E^{n}(0,j-1,k+½)
3A	D 3.2	D 3.2	E^{n}(1.j-1,k+½)
3C	D 4.6	D 4.6	E^{n}(0,j,k+½)
3E	D 1.1	D 1.1	E^{n}(1,j,k+½)
40	D 4.1	D 4.1	E^{n+1}(0,j,k+½)
42	D 2.1	D 2.1	E^{n-1}(0,j,k+½)
44	D 6.7	D 6.7	E^{n-1}(1,j,k+½)
46	D 4.7	D 4.7	E^{n}(0,j+1,k+½)
48	D 2.5	D 2.5	E^{n}(1,j+1,k+½)
4A	D 3.5	D 3.5	E^{n+1}(0,j+1,k+½)
4C	D 3.2	D 3.2	E^{n-1}(0,j+1,k+½)
4E	D 4.4	D 4.4	E^{n-1}(1,j+1,k+½)
50	D 5.4	D 5.4	E^{n}(0,j+2,k+½)
52	D 7.3	D 7.3	E^{n}(1,j+2,k+½)
54	D 4.3	D 4.3	E^{n+1}(0,j+2,k+½)
56	D 0.6	D 0.6	E^{n-1}(0,j+2,k+½)
58	D 0.2	D 0.2	E^{n-1}(1,j+2,k+½)
5A	D 0.5	D 0.5	E^{n}(0,j+3,k+½)
5C	D 4.5	D 4.5	E^{n}(1,j+3,k+½)

5E	H 00000000	H 00000000		
60	H 00000000	H 00000000		
62	H 00000000	H 00000000	{K1}	[Next data vector]
64	H 00000000	H 00000000	{K2}	
66	H 00000000	H 00000000	{K3}	
68	H 00000000	H 00000000	{UP,DOWN}	
6A	H 00000000	H 00000000	{(Blank),OUT}	
6C	H 00000000	H 00000000	{$E^{n}(0,j-1,k+1\frac{1}{2})$}	
6E	H 00000000	H 00000000	{$E^{n}(1.j-1,k+1\frac{1}{2})$}	
70	H 00000000	H 00000000	$E^{n}(0,j,k+1\frac{1}{2})$	[Address DOWN]
72	D 8.1	D 8.1	$E^{n}(0,j,k+1\frac{1}{2})$	
74	D 7.1	D 7.1		
76	H 00000000	H 00000000	{$E^{n+1}(0,j,k+1\frac{1}{2})$}	
78	H 00000000	H 00000000	{$E^{n-1}(0,j,k+1\frac{1}{2})$}	
7A	H 00000000	H 00000000	{$E^{n-1}(1,j,k+1\frac{1}{2})$}	
7C	D 4.1	D 4.1	$E^{n}(0,j+1,k+1\frac{1}{2})$	
7E	D 3.1	D 3.1	$E^{n}(0,j+1,k+1\frac{1}{2})$	
80	H 00000000	H 00000000	{$E^{n+1}(0,j+1,k+\frac{1}{2})$}	
82	H 00000000	H 00000000	{$E^{n-1}(0,j+1,k+\frac{1}{2})$}	
84	H 00000000	H 00000000	{$E^{n-1}(1,j+1,k+\frac{1}{2})$}	
86	D 1.1	D 1.1	$E^{n(}0,j+2,k+1\frac{1}{2})$	
88	D 2.1	D 2.1	$E^{n}(0,j+2,k+1\frac{1}{2})$	
8A	H 00000000	H 00000000	Expect first answer here (138 decimal)	
8C	H 00000000	H 00000000		
8E	H 00000000	H 00000000		
90	H 00000000	H 00000000		
92	H 00000000	H 00000000		
94	H 00000000	H 00000000	Expect second answer here (148 decimal)	
96	H 00000000	H 00000000		
98	H 00000000	H 00000000		
9A	H 00000000	H 00000000		
9C	H 00000000	H 00000000		
9E	H 00000000	H 00000000	Expect third answer here (158 decimal)	
A0	H 00000000	H 00000000		
A2	D 96.98	D 96.98	Expected Answer #1	
A4	D 33.64	D 33.64	#2	
A6	D 34.86	D 34.86	#3	

Appendix H -- Microcode Results

This printout of FPASP4_U0.DAT shows the correct answers at decimal addresses 138, 148, and 158. Note how they match the preplaced values at 162, 164, and 166. FPASP4_U0.DAT contains only the less significant 32 bits of the double-precision numbers and is not shown.

```
  0:    0000002E  00000000  00000000  00000000  00000000
 10:    00000000  00000000  00000000  00000000  00000000
 20:    00000000  405B2666  401E6666  00000000  00000000
 30:    00000000  40113333  400A6666  00000000  00000000
 40:    00000000  3FF4CCCC  40026666  3FF33333  3FF66666
 50:    3FE33333  00000016  00000000  40153333  40099999
 60:    40126666  3FF19999  40106666  4000CCCC  401ACCCC
 70:    4012CCCC  40040000  400C0000  40099999  40119999
 80:    40159999  401D3333  40113333  3FE33333  3FC99999
 90:    3FE00000  40120000  00000000  00000000  00000000
100:    00000000  00000000  00000000  00000000  00000000
110:    00000000  00000000  40203333  401C6666  00000000
120:    00000000  00000000  40106666  4008CCCC  00000000
130:    00000000  00000000  3FF19999  4000CCCC  40583EB8
140:    00000000  00000000  00000000  00000000  4040D1EB
150:    00000000  00000000  00000000  00000000  40416E14
160:    00000000  40583EB8  4040D1EB  40416E14  00000000
```

Bibliography

1. Katz, Daniel S. and Allen Taflove. "Large-scale Methods in Computational Electromagnetics," *Cray Channels*, 13: 16-19 (Spring 1991).

2. Yee, Kane S. "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*, 14: 302-307 (May 1966).

3. Child, Jeffrey. "256-kbit SRAMs provide many choices, while 1-Mbit chips gain speed," *Computer Design* (30: 99-105) April 1, 1990.

4. ANSI/IEEE Std 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., New York, 1985.

5. Perlik, Andrew T. and others. "Predicting Scattering of Electromagnetic Fields Using FD-TD on a Connection Machine," *IEEE Transactions on Magnetics*, 25: 2910-2912 (July 1989).

6. Calalo, R. H. and others. "Hypercube Parallel Architecture Applied to Electromagnetic Scattering Analysis," *IEEE Transactions on Magnetics*, 25: 2898-2900 (July 1989).

7. *The nCUBE 2 Supercomputers*. Sales Brochure 1002.1090. nCUBE Corporation, Beaverton, OR, 1990.

8. Corcoran, Mark, Sales Executive. Telephone Conversation. nCUBE Corporation, Cincinnati, OH, 17 Oct 1991.

9. Larson, Ronal W., and others. "Special Purpose Computers for the Time Domain Advance of Maxwell's Equations," *IEEE Transactions on Magnetics*, 25: 2913-2915 (July 1989).

10. *EM Wavetracer Version 1.0*. Sales Sheet. Wavetracer, Inc, Acton, MA, 01720.

11. Telephone interview, Wavetracer Inc., 289 Great Road, MA, 4 Oct 1991.

12. Taflove, Allen, and Korada R. Umashankar. "Review of FD-TD Numerical Modeling of Electromagnetic Wave Scattering and Radar Cross Section," *Proceedings of the IEEE*, 5: 682-699 (May 1989).

13. Balanis, Constantine A. *Advanced Engineering Electromagnetics*. New York: John Wiley and Sons, 1989.

14. Harrington, Roger F. Time-Harmonic Electromagnetic Fields. New York: McGraw-Hill Book Company, 1961.

15. Taflove, Allen, and Morris E. Brodwin. "Numerical Solution of Steady-State Electromagnetic Scattering Problems Using the Time-Dependent Maxwell's Equations," *IEEE Transactions on Microwave Theory and Techniques*, 23: 623-630 (August 1975).

16. Mur, Gerrit. "Absorbing boundary Conditions for the Finite-Difference Approximation of the Time-Domain Electromagnetic-Field Equations," *Transactions on Electromagnetic Compatibility*, 23: 377-382 (November 1981).

17. Fusco, Mario. "FDTD Algorithm in Curvilinear Coordinates," *IEEE Transactions on Antennas and Propagation*, 38: 76-89 (January 1990).

18. Harfoush, Fady, and others. "A Numerical Technique for Analyzing Electromagnetic Wave Scattering from Moving Surfaces in One and Two Dimensions," *IEEE Transactions on Antennas and Propagation*, 37: 55-63 (January 1989).

19. Luebbers, Raymond, and others. "A Frequency-Dependent Finite-Difference Time-Domain Formulation for Dispersive Materials," *IEEE Transactions on Electromagnetic Compatibility*, 32: 222-227 (August 1990).

20. Katz, Daniel S. and others. "FDTD Analysis of Electromagnetic Wave Radiation from Systems Containing Horn Antennas," *IEEE Transactions on Antennas and Propagation*, 39: 1203-1212 (August 1991).

21. Strauss, Capt Jack L. *Architectural Implications of a Parallel Computational Approach to the Vector Wave Equation*. MS Thesis, AFIT/GE/ENG/87M-5. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1987.

22. Comtois, Capt John Henry. *Architecture and Design for a Laser Programmable Double Precision Floating Point Application Specific Processor*. MS Thesis, AFIT/GE/ENG/88-5. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

23. WSVP User's Manual, Version 4.7 Rome Laboratory/OCTS.

24. Enriquez, Arnel B., and Keith R. Jones. *Design of a Multi-Mode Pipelined Multiplier for Floating Point Applications*. Unpublished report. February 5, 1991.

25. Luebbers, Raymond J. *Penn State University Finite Difference Time Domain Electromagnetic Analysis Computer Code -- Version D*. Penn State University, University Park, PA, 16802.

26. Hayes, John P. *Computer Architecture and Organization* (Second Edition). McGraw-Hill Book Company, 1988.

27. Hogberg, Dan, Account Manager, Telephone Interview, Cray Research, Inc., Minneapolis, MN, 17 Oct 1991.